NAME

intro, **errno** — introduction to system calls and error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

This section provides an overview of the system calls, their error returns, and other common definitions and concepts.

DIAGNOSTICS

Nearly all of the system calls provide an error number in the external variable *errno*. *errno* is implemented as a macro which expands to a modifiable lvalue of type *int*.

When a system call detects an error, it returns an integer value indicating failure (usually −1) and sets the variable *errno* accordingly. (This allows interpretation of the failure on receiving a −1 and to take action accordingly.) Successful calls never set *errno*; once set, it remains until another error occurs. It should only be examined after an error. Note that a number of system calls overload the meanings of these error numbers, and that the meanings must be interpreted according to the type and circumstances of the call.

The manual page for each system call will list some of the common errno codes that system call can return, but that should not be considered an exhaustive list, i.e. a properly written program should be able to gracefully recover from any error that a system call might return. Documenting all the error codes that a system call can return in a more specification-like manner would take more resources than this project has available.

The following is a complete list of the errors and their names as given in ⟨errno.h⟩.

0 *Error 0*. Not used.

1 EPERM *Operation not permitted*. An attempt was made to perform an operation limited to processes with appropriate privileges or to the owner of a file or other resources.

2 ENOENT *No such file or directory*. A component of a specified pathname did not exist, or the pathname was an empty string.

3 ESRCH *No such process*. No process could be found corresponding to that specified by the given process ID.

4 EINTR *Interrupted function call*. An asynchronous signal (such as SIGINT or SIGQUIT) was caught by the process during the execution of an interruptible function. If the signal handler performs a normal return, the interrupted function call will seem to have returned the error condition.

5 EIO *Input/output error*. Some physical input or output error occurred. This error will not be reported until a subsequent operation on the same file descriptor and may be lost (over written) by any subsequent errors.

6 ENXIO *Device not configured*. Input or output on a special file referred to a device that did not exist, or made a request beyond the limits of the device. This error may also occur when, for example, a tape drive is not online or no disk pack is loaded on a drive.

7 E2BIG *Arg list too long*. The number of bytes used for the argument and environment list of the new process exceeded the current limit of 2^{18} bytes (ARG_MAX in ⟨sys/syslimits.h⟩).

8 ENOEXEC *Exec format error*. A request was made to execute a file that, although it has the appropriate permissions, was not in the format required for an executable file.

9 `EBADF` *Bad file descriptor*. A file descriptor argument was out of range, referred to no open file, had been revoked by `revoke`(2), or a `read`(2) (or `write`(2)) request was made to a file that was only open for writing (or reading).

10 `ECHILD` *No child processes*. A `wait`(2) or `waitpid`(2) function was executed by a process that had no existing or unwaited-for child processes.

11 `EDEADLK` *Resource deadlock avoided*. An attempt was made to lock a system resource that would have resulted in a deadlock situation.

12 `ENOMEM` *Cannot allocate memory*. The new process image required more memory than was allowed by the hardware or by system-imposed memory management constraints. A lack of swap space is normally temporary; however, a lack of core is not. Soft limits may be increased to their corresponding hard limits.

13 `EACCES` *Permission denied*. An attempt was made to access a file in a way forbidden by its file access permissions.

14 `EFAULT` *Bad address*. The system detected an invalid address in attempting to use an argument of a call. The reliable detection of this error cannot be guaranteed and when not detected may result in the generation of a signal, indicating an address violation, which is sent to the process.

15 `ENOTBLK` *Block device required*. A block device operation was attempted on a non-block device or file.

16 `EBUSY` *Resource busy*. An attempt to use a system resource which was in use at the time in a manner which would have conflicted with the request.

17 `EEXIST` *File exists*. An existing file was mentioned in an inappropriate context, for instance, as the new link name in a `link`(2) function.

18 `EXDEV` *Improper link*. A hard link to a file on another file system was attempted.

19 `ENODEV` *Operation not supported by device*. An attempt was made to apply an inappropriate function to a device, for example, trying to read a write-only device such as a printer.

20 `ENOTDIR` *Not a directory*. A component of the specified pathname existed, but it was not a directory, when a directory was expected.

21 `EISDIR` *Is a directory*. An attempt was made to open a directory with write mode specified.

22 `EINVAL` *Invalid argument*. Some invalid argument was supplied. (For example, specifying an undefined signal to a `signal`(3) or `kill`(2) function).

23 `ENFILE` *Too many open files in system*. Maximum number of file descriptors allowable on the system has been reached and a requests for an open cannot be satisfied until at least one has been closed.

24 `EMFILE` *Too many open files*. <As released, the limit on the number of open files per process is 64.> The `getrlimit`(2) call with the *`RLIMIT_NOFILE`* resource will obtain the current limit.

25 `ENOTTY` *Inappropriate ioctl for device*. A control function (see `ioctl`(2)) was attempted for a file or special device for which the operation was inappropriate.

26 `ETXTBSY` *Text file busy*. The new process was a pure procedure (shared text) file which was open for writing by another process, or while the pure procedure file was being executed an `open`(2) call requested write access.

27 `EFBIG` *File too large*. The size of a file exceeded the maximum. (The system-wide maximum file size is 2^{63} bytes. Each file system may impose a lower limit for files contained within it).

28 `ENOSPC` *Device out of space*. A `write`(2) to an ordinary file, the creation of a directory or symbolic link, or the creation of a directory entry failed because no more disk blocks were available on the file system, or the allocation of an inode for a newly created file failed because no more inodes were available on the file system.

29 `ESPIPE` *Illegal seek*. An `lseek`(2) function was issued on a socket, pipe or FIFO.

30 `EROFS` *Read-only file system*. An attempt was made to modify a file or directory was made on a file system that was read-only at the time.

31 `EMLINK` *Too many links*. The number of hard links to a single file has exceeded the maximum. (The system-wide maximum number of hard links is 32767. Each file system may impose a lower limit for files contained within it).

32 `EPIPE` *Broken pipe*. A write on a pipe, socket or FIFO for which there is no process to read the data.

33 `EDOM` *Numerical argument out of domain*. A numerical input argument was outside the defined domain of the mathematical function.

34 `ERANGE` *Result too large or too small*. The result of the function is too large or too small to be represented in the available space.

35 `EAGAIN` *Resource temporarily unavailable*. This is a temporary condition and later calls to the same routine may complete normally.

36 `EINPROGRESS` *Operation now in progress*. An operation that takes a long time to complete (such as a `connect`(2)) was attempted on a non-blocking object (see `fcntl`(2)).

37 `EALREADY` *Operation already in progress*. An operation was attempted on a non-blocking object that already had an operation in progress.

38 `ENOTSOCK` *Socket operation on non-socket*. Self-explanatory.

39 `EDESTADDRREQ` *Destination address required*. A required address was omitted from an operation on a socket.

40 `EMSGSIZE` *Message too long*. A message sent on a socket was larger than the internal message buffer or some other network limit.

41 `EPROTOTYPE` *Protocol wrong type for socket*. A protocol was specified that does not support the semantics of the socket type requested. For example, you cannot use the ARPA Internet UDP protocol with type `SOCK_STREAM`.

42 `ENOPROTOOPT` *Protocol option not available*. A bad option or level was specified in a `getsockopt`(2) or `setsockopt`(2) call.

43 `EPROTONOSUPPORT` *Protocol not supported*. The protocol has not been configured into the system or no implementation for it exists.

44 `ESOCKTNOSUPPORT` *Socket type not supported*. The support for the socket type has not been configured into the system or no implementation for it exists.

45 `EOPNOTSUPP` *Operation not supported*. The attempted operation is not supported for the type of object referenced. Usually this occurs when a file descriptor refers to a file or socket that cannot support this operation, for example, trying to *accept* a connection on a datagram socket.

46 `EPFNOSUPPORT` *Protocol family not supported*. The protocol family has not been configured into the system or no implementation for it exists.

47 `EAFNOSUPPORT` *Address family not supported by protocol family*. An address incompatible with the requested protocol was used. For example, you shouldn't necessarily expect to be able to use NS addresses with ARPA Internet protocols.

48 `EADDRINUSE` *Address already in use*. Only one usage of each address is normally permitted.

49 `EADDRNOTAVAIL` *Cannot assign requested address*. Normally results from an attempt to create a socket with an address not on this machine.

50 `ENETDOWN` *Network is down*. A socket operation encountered a dead network.

51 `ENETUNREACH` *Network is unreachable*. A socket operation was attempted to an unreachable network.

52 `ENETRESET` *Network dropped connection on reset*. The host you were connected to crashed and rebooted.

53 `ECONNABORTED` *Software caused connection abort*. A connection abort was caused internal to your host machine.

54 `ECONNRESET` *Connection reset by peer*. A connection was forcibly closed by a peer. This normally results from a loss of the connection on the remote socket due to a timeout or a reboot.

55 `ENOBUFS` *No buffer space available*. An operation on a socket or pipe was not performed because the system lacked sufficient buffer space or because a queue was full.

56 `EISCONN` *Socket is already connected*. A `connect`(2) request was made on an already connected socket; or, a `sendto`(2) or `sendmsg`(2) request on a connected socket specified a destination when already connected.

57 `ENOTCONN` *Socket is not connected*. An request to send or receive data was disallowed because the socket was not connected and (when sending on a datagram socket) no address was supplied.

58 `ESHUTDOWN` *Cannot send after socket shutdown*. A request to send data was disallowed because the socket had already been shut down with a previous `shutdown`(2) call.

60 `ETIMEDOUT` *Operation timed out*. A `connect`(2) or `send`(2) request failed because the connected party did not properly respond after a period of time. (The timeout period is dependent on the communication protocol).

61 `ECONNREFUSED` *Connection refused*. No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host.

62 `ELOOP` *Too many levels of symbolic links*. A path name lookup involved more than 32 (`MAXSYMLINKS`) symbolic links.

63 `ENAMETOOLONG` *File name too long*. A component of a path name exceeded 255 (`MAXNAMELEN`) characters, or an entire path name exceeded 1023 (`MAXPATHLEN−1`) characters.

64 `EHOSTDOWN` *Host is down*. A socket operation failed because the destination host was down.

65 `EHOSTUNREACH` *No route to host*. A socket operation was attempted to an unreachable host.

66 `ENOTEMPTY` *Directory not empty*. A directory with entries other than '`.`' and '`..`' was supplied to a remove directory or rename call.

67 `EPROCLIM` *Too many processes*.

68 `EUSERS` *Too many users*. The quota system ran out of table entries.

69 `EDQUOT` *Disc quota exceeded*. A `write`(2) to an ordinary file, the creation of a directory or symbolic link, or the creation of a directory entry failed because the user's quota of disk blocks was exhausted, or the allocation of an inode for a newly created file failed because the user's quota of inodes was exhausted.

70 `ESTALE` *Stale NFS file handle*. An attempt was made to access an open file (on an NFS filesystem) which is now unavailable as referenced by the file descriptor. This may indicate the file was deleted on the NFS server or some other catastrophic event occurred.

72 `EBADRPC` *RPC struct is bad*. Exchange of RPC information was unsuccessful.

73 `ERPCMISMATCH` *RPC version wrong*. The version of RPC on the remote peer is not compatible with the local version.

74 `EPROGUNAVAIL` *RPC prog. not avail*. The requested program is not registered on the remote host.

75 `EPROGMISMATCH` *Program version wrong*. The requested version of the program is not available on the remote host (RPC).

76 `EPROCUNAVAIL` *Bad procedure for program*. An RPC call was attempted for a procedure which doesn't exist in the remote program.

77 `ENOLCK` *No locks available*. A system-imposed limit on the number of simultaneous file locks was reached.

78 `ENOSYS` *Function not implemented*. Attempted a system call that is not available on this system.

79 `EFTYPE` *Inappropriate file type or format*. Attempted a file operation on a file of a type for which it was invalid.

80 `EAUTH` *Authentication error*. Attempted to use an invalid authentication ticket to mount an NFS filesystem.

81 `ENEEDAUTH` *Need authenticator*. An authentication ticket must be obtained before the given NFS filesystem may be mounted.

82 `EIDRM` *Identifier removed*. An IPC identifier was removed while the current process was waiting on it.

83 `ENOMSG` *No message of the desired type*. An IPC message queue does not contain a message of the desired type, or a message catalog does not contain the requested message.

84 `EOVERFLOW` *Value too large to be stored in data type*. A numerical result of the function was too large to be stored in the caller-provided space.

85 `EILSEQ` *Illegal byte sequence*. A wide character/multibyte character encoding error occurred.

86 `ENOTSUP` *Not supported*. An attempt was made to set or change a parameter to an unsupported value.

87 `ECANCELED` *Operation canceled*. The requested operation was canceled.

88 `EBADMSG` *Bad or corrupt message*. A message in the specified message catalog did not satisfy implementation defined criteria, or a STREAMS operation encountered an invalid message or a file descriptor at the STREAM head.

89 `ENODATA` *No message available*. No message is available on the STREAM head read queue

90 `ENOSR` *No STREAM resources*. Buffers could not be allocated due to insufficient STREAMs memory resources.

91 `ENOSTR` *Not a STREAM*. A STREAM is not associated with the specified file descriptor.

`92 ETIME` *STREAM ioctl timeout*. The timer set for a STREAMS `ioctl`(2) operation has expired.

`93 ENOATTR` *Attribute not found*. The specified extended attribute does not exist.

`94 EMULTIHOP` *Multihop attempted*. Components of path require hopping to multiple remote machines and the file system does not allow it. It occurs when users try to access remote resources which are not directly accessible.

`95 ENOLINK` *Link has been severed*. Occurs when the link (virtual circuit) connecting to a remote machine is gone.

`96 EPROTO` *Protocol error*. Some protocol error occurred. This error is device-specific, but is generally not related to a hardware failure.

DEFINITIONS

Process ID
: Each active process in the system is uniquely identified by a non-negative integer called a process ID. The range of this ID is from 0 to 30000.

Parent process ID
: A new process is created by a currently active process; (see `fork`(2)). The parent process ID of a process is initially the process ID of its creator. If the creating process exits, the parent process ID of each child is set to the ID of a system process, `init`(8).

Process Group
: Each active process is a member of a process group that is identified by a non-negative integer called the process group ID. This is the process ID of the group leader. This grouping permits the signaling of related processes (see `termios`(4)) and the job control mechanisms of `csh`(1).

Session
: A session is a set of one or more process groups. A session is created by a successful call to `setsid`(2), which causes the caller to become the only member of the only process group in the new session.

Session leader
: A process that has created a new session by a successful call to `setsid`(2), is known as a session leader. Only a session leader may acquire a terminal as its controlling terminal (see `termios`(4)).

Controlling process
: A session leader with a controlling terminal is a controlling process.

Controlling terminal
: A terminal that is associated with a session is known as the controlling terminal for that session and its members.

Terminal Process Group ID
: A terminal may be acquired by a session leader as its controlling terminal. Once a terminal is associated with a session, any of the process groups within the session may be placed into the foreground by setting the terminal process group ID to the ID of the process group. This facility is used to arbitrate between multiple jobs contending for the same terminal. (See `csh`(1) and `tty`(4) for more information on job control.)

Orphaned Process Group
: A process group is considered to be *orphaned* if it is not under the control of a job control shell. More precisely, a process group is orphaned when none of its members has a parent process that is in the same session as the group, but is in a different process group. Note that when a process exits, the parent process for its children is changed to be `init`(8), which is in a separate session. Not all members of an orphaned process group are necessarily orphaned processes (those whose creating

process has exited). The process group of a session leader is orphaned by definition.

Real User ID and Real Group ID

Each user on the system is identified by a positive integer termed the real user ID.

Each user is also a member of one or more groups. One of these groups is distinguished from others and used in implementing accounting facilities. The positive integer corresponding to this distinguished group is termed the real group ID.

All processes have a real user ID and real group ID. These are initialized from the equivalent attributes of the process that created it.

Effective User Id, Effective Group Id, and Group Access List

Access to system resources is governed by two values: the effective user ID, and the group access list. The first member of the group access list is also known as the effective group ID. (In POSIX.1, the group access list is known as the set of supplementary group IDs, and it is unspecified whether the effective group ID is a member of the list.)

The effective user ID and effective group ID are initially the process's real user ID and real group ID respectively. Either may be modified through execution of a set-user-ID or set-group-ID file (possibly by one its ancestors) (see `execve`(2)). By convention, the effective group ID (the first member of the group access list) is duplicated, so that the execution of a set-group-ID program does not result in the loss of the original (real) group ID.

The group access list is a set of group IDs used only in determining resource accessibility. Access checks are performed as described below in "File Access Permissions".

Saved Set User ID and Saved Set Group ID

When a process executes a new file, the effective user ID is set to the owner of the file if the file is set-user-ID, and the effective group ID (first element of the group access list) is set to the group of the file if the file is set-group-ID. The effective user ID of the process is then recorded as the saved set-user-ID, and the effective group ID of the process is recorded as the saved set-group-ID. These values may be used to regain those values as the effective user or group ID after reverting to the real ID (see `setuid`(2)). (In POSIX.1, the saved set-user-ID and saved set-group-ID are optional, and are used in setuid and setgid, but this does not work as desired for the super-user.)

Super-user

A process is recognized as a *super-user* process and is granted special privileges if its effective user ID is 0.

Special Processes

The processes with process IDs of 0 and 1 are special. Process 0 is the scheduler. Process 1 is the initialization process `init`(8), and is the ancestor (parent) of every other process in the system. It is used to control the process structure. The kernel will allocate other kernel threads to handle certain periodic tasks or device related tasks, such as:

acctwatch	System accounting disk watcher, see `acct`(2), `acct`(5).
aiodoned	Asynchronous I/O done handler, see `uvm`(9).
atabusX	ATA bus handler, see `ata`(4).
cardslotX	CardBus slot watcher thread, see `cardslot`(4).
cryptoret	The software crypto daemon.
fssbsX	File system snapshot thread, see `fss`(4).

ioflush The in-kernel periodic flush the buffer cache to disk task, which replaces the old **update** program.

nfsio, **nfskqpoll**
NFS handing daemons.

lfs_writer Log filesystem writer.

pagedaemon The page daemon.

raidX, **raidioX**, **raid_parity**, **raid_recon**, **raid_reconip**, **raid_copyback**
Raid framework related threads, see `raid`(4).

scsibusX SCSI bus handler, see `scsi`(4).

smbiodX, **smbkq**
SMBFS handling daemon, see `netsmb`(4).

swdmover The software data mover I/O thread, see `dmoverio`(4).

sysmon The systems monitoring framework daemon.

usbX, **usbtask**
USB bus handler, see `usb`(4).

There are more machine-dependent kernel threads allocated by different drivers. See the specific driver manual pages for more information.

Descriptor
An integer assigned by the system when a file is referenced by `open`(2) or `dup`(2), or when a socket is created by `pipe`(2), `socket`(2), or `socketpair`(2), which uniquely identifies an access path to that file or socket from a given process or any of its children.

File Name
Names consisting of up to 255 (`MAXNAMELEN`) characters may be used to name an ordinary file, special file, or directory.

These characters may be selected from the set of all ASCII character excluding 0 (NUL) and the ASCII code for '`/`' (slash). (The parity bit, bit 7, must be 0).

Note that it is generally unwise to use '`*`', '`?`', '`[`' or '`]`' as part of file names because of the special meaning attached to these characters by the shell.

Pathname
A path name is a NUL-terminated character string starting with an optional slash '`/`', followed by zero or more directory names separated by slashes, optionally followed by a file name. The total length of a path name must be less than 1024 (`MAXPATHLEN`) characters.

If a path name begins with a slash, the path search begins at the *root* directory. Otherwise, the search begins from the current working directory. A slash by itself names the root directory. An empty string is not a valid pathname.

Directory
A directory is a special type of file that contains entries that are references to other files. Directory entries are called links. By convention, a directory contains at least two links, '`.`' and '`..`', referred to as *dot* and *dot-dot* respectively. Dot refers to the directory itself and dot-dot refers to its parent directory.

Root Directory and Current Working Directory
Each process has associated with it a concept of a root directory and a current working directory for the purpose of resolving path name searches. A process's root directory need not be the root direc-

tory of the root file system.

File Access Permissions

Every file in the file system has a set of access permissions. These permissions are used in determining whether a process may perform a requested operation on the file (such as opening a file for writing). Access permissions are established at the time a file is created. They may be changed at some later time through the `chmod`(2) call.

File access is broken down according to whether a file may be: read, written, or executed. Directory files use the execute permission to control if the directory may be searched.

File access permissions are interpreted by the system as they apply to three different classes of users: the owner of the file, those users in the file's group, anyone else. Every file has an independent set of access permissions for each of these classes. When an access check is made, the system decides if permission should be granted by checking the access information applicable to the caller.

Read, write, and execute/search permissions on a file are granted to a process if:

The process's effective user ID is that of the super-user. (Note: even the super-user cannot execute a non-executable file).

The process's effective user ID matches the user ID of the owner of the file and the owner permissions allow the access.

The process's effective user ID does not match the user ID of the owner of the file, and either the process's effective group ID matches the group ID of the file, or the group ID of the file is in the process's group access list, and the group permissions allow the access.

Neither the effective user ID nor effective group ID and group access list of the process match the corresponding user ID and group ID of the file, but the permissions for ‘‘other users’’ allow access.

Otherwise, permission is denied.

Sockets and Address Families

A socket is an endpoint for communication between processes. Each socket has queues for sending and receiving data.

Sockets are typed according to their communications properties. These properties include whether messages sent and received at a socket require the name of the partner, whether communication is reliable, the format used in naming message recipients, etc.

Each instance of the system supports some collection of socket types; consult `socket`(2) for more information about the types available and their properties.

Each instance of the system supports some number of sets of communications protocols. Each protocol set supports addresses of a certain format. An Address Family is the set of addresses for a specific group of protocols. Each socket has an address chosen from the address family in which the socket was created.

SEE ALSO

`intro`(3), `perror`(3)

HISTORY

An **`intro`** manual page appeared in Version 6 AT&T UNIX.

NAME

_Exit, _exit — terminate the calling process

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <stdlib.h>

void
_Exit(*int status*);

#include <unistd.h>

void
_exit(*int status*);

DESCRIPTION

The **_Exit**() and **_exit**() functions are equivalent. They each terminate a process with the following consequences:

- All of the descriptors open in the calling process are closed. This may entail delays, for example, waiting for output to drain; a process in this state may not be killed, as it is already dying.
- If the parent process of the calling process has an outstanding `wait`(2) call or catches the `SIGCHLD` signal, it is notified of the calling process's termination and the *status* is set as defined by `wait`(2).
- The parent process-ID of all of the calling process's existing child processes are set to 1; the initialization process (see the DEFINITIONS section of `intro`(2)) inherits each of these processes.
- If the termination of the process causes any process group to become orphaned (usually because the parents of all members of the group have now exited; see "orphaned process group" in `intro`(2)), and if any member of the orphaned group is stopped, the `SIGHUP` signal and the `SIGCONT` signal are sent to all members of the newly-orphaned process group.
- If the process is a controlling process (see `intro`(2)), the `SIGHUP` signal is sent to the foreground process group of the controlling terminal, and all current access to the controlling terminal is revoked.

Most C programs call the library routine `exit`(3), which flushes buffers, closes streams, unlinks temporary files, etc., before calling **_exit**().

RETURN VALUES

_Exit() and **_exit**() can never return.

SEE ALSO

`fork`(2), `sigaction`(2), `wait`(2), `exit`(3)

STANDARDS

The **_exit**() function conforms to ISO/IEC 9945-1:1990 ("POSIX.1"). The **_Exit**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

NAME

_lwp_create — create a new light-weight process

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <lwp.h>

int
_lwp_create(*ucontext_t *context*, *unsigned long flags*, *lwpid_t *new_lwp*);

DESCRIPTION

_lwp_create() causes creation of a new light-weight process, or LWP, and adds it to the current process. The *context* argument specifies the initial execution context for the new LWP including signal mask, stack, and machine registers.

The following flags affect the creation of the new LWP:

LWP_DETACHED	The LWP is created detached. The resources associated with a detached LWP will be automatically reclaimed by the system when the LWP exits. Otherwise, a terminated LWP's resources will not be reclaimed until its status is reported to another LWP via _lwp_wait(2).
LWP_SUSPENDED	The LWP is created suspended, and will not begin execution until it is resumed by another LWP via _lwp_continue(2).

The LWP ID of the new LWP is stored in the location pointed to by *new_lwp*.

RETURN VALUES

Upon successful completion, **_lwp_create**() returns a value of 0. Otherwise, an error code is returned to indicate the error.

ERRORS

_lwp_create() will fail and no LWP will be created if:

[EAGAIN]	The system-imposed limit on the total number of LWPs under execution would be exceeded. This limit is configuration-dependent.
[ENOMEM]	There is insufficient swap space for the new LWP.
[EFAULT]	The address pointed to by *context* or *new_lwp* is outside the process's allocated address space.

SEE ALSO

_lwp_continue(2), _lwp_exit(2), _lwp_wait(2), _lwp_makecontext(3)

HISTORY

The **_lwp_create**() system call first appeared in NetBSD 2.0.

NAME

`_lwp_ctl` — prepare per-LWP communication area between kernel and userland

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

`#include <lwp.h>`

`int`
`_lwp_ctl`(*`int features`*, *`struct lwpctl **address`*);

DESCRIPTION

`_lwp_ctl`() prepares per-LWP communication area for the calling LWP, and maps it into the calling process' address space. It takes the following arguments.

`features` The bitwise-OR of the following flags.

`LWPCTL_FEATURE_CURCPU`	Request lc_curcpu.
`LWPCTL_FEATURE_PCTR`	Request lc_pctr.

`address` The address to store a pointer to lwpctl structure for the calling LWP.

The per-LWP communication area is described by an lwpctl structure. It has following members, depending on *`features`*.

int lc_curcpu	The integral identifier of the CPU on which the LWP is running, or `LWPCTL_CPU_NONE` when the LWP is not running on any CPU. It's updated by the kernel and should be considered as read-only for userland. It's available only if requested with the `LWPCTL_FEATURE_CURCPU` flag.
int lc_pctr	The integer which is incremented on every context switches to the LWP. It can be used to detect preemption of the LWP. (thus its name "preemption counter".) It's updated by the kernel and should be considered as read-only for userland. It's available only if requested with the `LWPCTL_FEATURE_PCTR` flag.

RETURN VALUES

`_lwp_ctl`() returns 0 on success. Otherwise, −1 is returned and *errno* is set to indicate the error.

SEE ALSO

`errno`(2)

NAME

_lwp_detach — detach a light-weight process

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <lwp.h>

int
_lwp_detach(*lwpid_t lwp*);

DESCRIPTION

_lwp_detach() causes a light-weight process to become detached, having the same effect as if the LWP was created with the LWP_DETACHED flag.

The resources associated with a detached LWP will be automatically reclaimed by the system when the LWP exits. Conversely, an attached LWP's resources will not be reclaimed until its status is reported to another LWP via _lwp_wait(2).

RETURN VALUES

A 0 value indicates that the call succeeded. A −1 return value indicates an error occurred and *errno* is set to indicate the reason.

ERRORS

[EINVAL]	The LWP is already detached.
[ESRCH]	No LWP can be found in the current process corresponding to that specified by *lwp*.

SEE ALSO

_lwp_create(2), _lwp_wait(2)

HISTORY

The **_lwp_detach**() system call first appeared in NetBSD 5.0.

NAME

_lwp_exit — terminate the calling light-weight process

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <lwp.h>

void
_lwp_exit(*void*);

DESCRIPTION

_lwp_exit() terminates the calling LWP. If it is the last LWP in the process, the process exits with a status of 0. If the LWP was not created in a detached state, then the system will not reclaim its LWP ID until its status is reported to another LWP in the process via _lwp_wait(2).

RETURN VALUES

_lwp_exit() can never return.

SEE ALSO

_exit(2), _lwp_create(2), _lwp_wait(2)

HISTORY

The **_lwp_exit**() system call first appeared in NetBSD 2.0.

NAME

_lwp_getname — get descriptive name of an LWP

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <lwp.h>

int
_lwp_getname(*lwpid_t target*, *char *name*, *size_t len*);

DESCRIPTION

_lwp_getname() gets the descriptive name of the LWP. It takes the following arguments.

target	The LWP whose descriptive name will be obtained.
name	The buffer to be filled with the descriptive name of the LWP.
len	The size of the buffer *name* in bytes.

RETURN VALUES

_lwp_getname() returns 0 on success. Otherwise, −1 is returned and *errno* is set to indicate the error.

SEE ALSO

top(1), ps(1), errno(2), _lwp_setname(2)

NAME

_lwp_getprivate, **_lwp_setprivate** — get and set light-weight process private data

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <lwp.h>

void *
_lwp_getprivate(*void*);

void
_lwp_setprivate(*void *ptr*);

DESCRIPTION

_lwp_setprivate() stores the pointer to private data *ptr* in a location private to the LWP.

_lwp_getprivate() returns the pointer to private data for the LWP.

ERRORS

The **_lwp_getprivate**() and **_lwp_setprivate**() functions are always successful, and no return value is reserved to indicate an error.

SEE ALSO

_lwp_makecontext(3)

HISTORY

The **_lwp_getprivate**() and **_lwp_setprivate**() system calls first appeared in NetBSD 2.0.

NAME

_lwp_kill — send a signal to a light-weight process

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <lwp.h>

int
_lwp_kill(*lwpid_t lwp*, *int sig*);

DESCRIPTION

_lwp_kill() sends the signal specified by *sig* to the light-weight process specified by *lwp*. If the *sig* argument is given as 0 (zero), **_lwp_kill** will test for the existance of the target LWP, but will take no further action.

Job control signals and uncatchable signals can not be directed to a specific LWP: if posted with **_lwp_kill**, they will affect all LWPs in the process.

Signals will be posted successfully to suspended LWPs, but will not be handled further until the LWP has been continued.

RETURN VALUES

A 0 value indicates that the call succeeded. A −1 return value indicates an error occurred and *errno* is set to indicate the reason.

ERRORS

[EINVAL]	*sig* is not a valid signal number.
[ESRCH]	No LWP can be found in the current process corresponding to that specified by *lwp*.

SEE ALSO

_lwp_continue(2), _lwp_suspend(2), kill(2), sigaction(2), signal(7)

HISTORY

The **_lwp_kill**() system call first appeared in NetBSD 5.0.

NAME

_lwp_park — wait interruptably in the kernel

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <lwp.h>

int
_lwp_park(*const struct timespec *abstime*, *lwpid_t unpark*, *const void *hint*, *const void *unparkhint*);

DESCRIPTION

_lwp_park() can be used to synchronize access to resources among multiple light-weight processes. It causes the calling LWP to wait interruptably in the kernel, until one of the following conditions is met:

- The *abstime* argument is non-NULL, and the absolute UTC time it specifies has passed.
- The LWP receives a directed signal posted using **_lwp_kill**(), or is elected to handle a signal on behalf of its containing process.
- The LWP is awoken by another LWP in the same process that has made a call to **_lwp_wakeup**().
- The LWP is awoken by another LWP in the same process that has made a call to **_lwp_unpark**() or **_lwp_unpark_all**().

The preferred method to awaken an LWP sleeping as a result of a call to **_lwp_park**() is to make a call to **_lwp_unpark**(), or **_lwp_unpark_all**(). The **_lwp_wakeup**() system call is a more general facility, and requires more resources to execute.

The optional *hint* argument specifies the address of object upon which the LWP is synchronizing. When the *hint* value is matched between calls to **_lwp_park**() and **_lwp_unpark**() or **_lwp_unpark_all**(), it may reduce the time necessary for the system to resume execution of waiting LWPs.

The *unpark* and *unparkhint* arguments can be used to fold a park operation and unpark operation into a single system call. If *unpark* is non-zero, the system will behave as if the following call had been made before the calling thread begins to wait:

```
      _lwp_unpark(unpark, unparkhint);
```

RETURN VALUES

_lwp_park() may return a value of 0. Otherwise, −1 is returned and *errno* is set to provide more information.

ERRORS

[EALREADY]
A request was made to wake the LWP before it began to wait in the kernel.

[EINTR] The LWP has been awoken by a signal or by a call to one of the following functions: **_lwp_unpark**(), **_lwp_unpark_all**(), **_lwp_wakeup**().

[EINVAL] The time value specified by *abstime* is invalid.

[ESRCH] No LWP can be found in the current process corresponding to *unpark*.

[ETIMEDOUT]
The UTC time specified by *abstime* has passed.

SEE ALSO

_lwp_unpark(2), _lwp_unpark_all(2), _lwp_wakeup(2)

HISTORY

The **_lwp_park**() system call first appeared in NetBSD 5.0.

NAME

_lwp_self — get light-weight process identification

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <lwp.h>

lwpid_t
_lwp_self(*void*);

DESCRIPTION

_lwp_self() returns the LWP ID of the calling LWP.

ERRORS

The **_lwp_self**() function is always successful, and no return value is reserved to indicate an error.

SEE ALSO

_lwp_create(2)

HISTORY

The **_lwp_self**() system call first appeared in NetBSD 2.0.

NAME

_lwp_setname — set descriptive name of an LWP

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <lwp.h>

int
_lwp_setname(*lwpid_t target*, *const char *name*);

DESCRIPTION

_lwp_setname() sets the descriptive name of the LWP. It takes the following arguments.

target	The LWP whose descriptive name will be set.
name	The string to be used as the descriptive name of the LWP.

The name is used by top(1) when showing LWPs, for example.

RETURN VALUES

_lwp_setname() returns 0 on success. Otherwise, −1 is returned and *errno* is set to indicate the error.

SEE ALSO

top(1), ps(1), errno(2), _lwp_getname(2)

NAME

_lwp_suspend, **_lwp_continue** — suspend or continue a light-weight process

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <lwp.h>

int
_lwp_suspend(*lwpid_t lwp*);

int
_lwp_continue(*lwpid_t lwp*);

DESCRIPTION

_lwp_suspend() suspends execution of the LWP specified by *lwp*. Once an LWP is suspended, subsequent calls to **_lwp_suspend**() have no effect. The only way to resume execution of a suspended LWP is via **_lwp_continue**().

_lwp_continue() resumes execution of the LWP specified by *lwp*. Once an LWP is resumed, subsequent calls to **_lwp_continue**() have no effect.

RETURN VALUES

Upon successful completion, **_lwp_suspend**() and **_lwp_continue**() return a value of 0. Otherwise, an error code is returned to indicate the error.

ERRORS

_lwp_suspend() and **_lwp_continue**() will fail if:

[ESRCH] No LWP can be found in the current process corresponding to that specified by *lwp*.

_lwp_suspend() will fail if:

[EDEADLK] The LWP specified by *lwp* is the only LWP in the process.

SEE ALSO

_lwp_create(2)

HISTORY

The **_lwp_create**() system call first appeared in NetBSD 2.0.

NAME

_lwp_unpark — resume execution of a waiting LWP

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <lwp.h>

int
_lwp_unpark(*lwpid_t lwp*, *const void *hint*);

DESCRIPTION

_lwp_unpark() resumes execution of the light-weight process *lwp*.

The target LWP is assumed to be waiting in the kernel as a result of a call to **_lwp_park**(). If the target LWP is not currently waiting, it will return immediatley upon the next call to **_lwp_park**().

See _lwp_park(2) for a description of the *hint* argument.

RETURN VALUES

A 0 value indicates that the call succeeded. A −1 return value indicates an error occurred and *errno* is set to indicate the reason.

ERRORS

[ESRCH]	No LWP can be found in the current process corresponding to that specified by *lwp*.

SEE ALSO

_lwp_park(2), _lwp_unpark_all(2), _lwp_wakeup(2)

HISTORY

The **_lwp_unpark**() system call first appeared in NetBSD 5.0.

NAME

_lwp_unpark_all — resume execution of a waiting LWP

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <lwp.h>

ssize_t
_lwp_unpark_all(*lwpid_t *targets*, *size_t ntargets*, *const void *hint*);

DESCRIPTION

_lwp_unpark_all() resumes execution of one or more light-weight processes listed in the array pointed to by *targets*.

The target LWPs are assumed to be waiting in the kernel as a result of calls to **_lwp_park**(). If any of the target LWPs are not currently waiting, those LWPs will return immediatley upon the next call to **_lwp_park**().

The value pointed to by *ntargets* specifies the size of the array pointed to by *targets*. If the *targets* argument is given as NULL, the maximum size of the array (expressed as the number of entries) is returned.

See _lwp_park(2) for a description of the *hint* argument.

RETURN VALUES

If the maximum size of the *targets* array is not being queried, a return of 0 indicates that the call succeeded. A −1 return value indicates an error occurred and *errno* is set to indicate the reason.

ERRORS

[EFAULT]	The value specified for *targets* is invalid.
[EINVAL]	The value specified for *ntargets* is out of range.
[ENOMEM]	Insufficient resources are available to complete the operation.

SEE ALSO

_lwp_park(2), _lwp_unpark(2), _lwp_wakeup(2)

HISTORY

The **_lwp_unpark_all**() system call first appeared in NetBSD 5.0.

NAME

_lwp_wait — wait for light-weight process termination

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <lwp.h>

int
_lwp_wait(*lwpid_t wlwp*, *lwpid_t *rlwp*);

DESCRIPTION

_lwp_wait() suspends execution of the calling LWP until the LWP specified by *wlwp* terminates. The specified LWP must not be detached. If *wlwp* is 0, then **_lwp_wait**() waits for any undetached LWP in the current process.

If *rlwp* is not NULL, then it points to the location where the LWP ID of the exited LWP is stored.

RETURN VALUES

Upon successful completion, **_lwp_wait**() returns a value of 0. Otherwise, an error code is returned to indicate the error.

ERRORS

_lwp_wait() will fail if:

[ESRCH]	No undetached LWP can be found in the current process corresponding to that specified by *wlwp*.
[EDEADLK]	The calling LWP is the only LWP in the process.
[EDEADLK]	The LWP ID specified by *wlwp* is the LWP ID of the calling LWP.
[EINTR]	**_lwp_wait**() was interrupted by a caught signal, or the signal did not have the SA_RESTART flag set.

SEE ALSO

_lwp_create(2), _lwp_exit(2)

HISTORY

The **_lwp_wait**() system call first appeared in NetBSD 2.0.

NAME

_lwp_wakeup — make a blocked light-weight process runnable

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <lwp.h>

int
_lwp_wakeup(*lwpid_t lwp*);

DESCRIPTION

_lwp_wakeup() makes a blocked LWP runnable. The blocked LWP must be in LSSLEEP state. Unblocking the LWP does not guarantee that it will make progress; it may block again as soon as it resumes execution in the kernel.

RETURN VALUES

Upon successful completion, **_lwp_wakeup**() returns a value of 0. Otherwise, an error code is returned to indicate the error.

ERRORS

_lwp_wakeup() will fail if:

[ESRCH]	No LWP can be found in the current process corresponding to that specified by *lwp*.
[ENODEV]	The specified LWP is not in LSSLEEP state.
[EBUSY]	The specified LWP is blocked in an uninterruptible sleep.

HISTORY

The **_lwp_wakeup**() system call first appeared in NetBSD 2.0.

NAME

accept — accept a connection on a socket

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/socket.h>

int
accept(*int s*, *struct sockaddr * restrict addr*,
    *socklen_t * restrict addrlen*);

DESCRIPTION

The argument *s* is a socket that has been created with socket(2), bound to an address with bind(2), and is listening for connections after a listen(2). The **accept**() argument extracts the first connection request on the queue of pending connections, creates a new socket with the same properties of *s* and allocates a new file descriptor for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, **accept**() blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, **accept**() returns an error as described below. The accepted socket may not be used to accept more connections. The original socket *s* remains open.

The argument *addr* is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the domain in which the communication is occurring. The *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*; on return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with SOCK_STREAM.

It is possible to select(2) or poll(2) a socket for the purposes of doing an **accept**() by selecting or polling it for read.

For certain protocols which require an explicit confirmation, such as ISO or DATAKIT, **accept**() can be thought of as merely dequeuing the next connection request and not implying confirmation. Confirmation can be implied by a normal read or write on the new file descriptor, and rejection can be implied by closing the new socket.

One can obtain user connection request data without confirming the connection by issuing a recvmsg(2) call with an *msg_iovlen* of 0 and a non-zero *msg_controllen*, or by issuing a getsockopt(2) request. Similarly, one can provide user connection rejection information by issuing a sendmsg(2) call with providing only the control information, or by calling setsockopt(2).

RETURN VALUES

The call returns −1 on error. If it succeeds, it returns a non-negative integer that is a descriptor for the accepted socket.

ERRORS

The **accept**() will fail if:

[EAGAIN]	The socket is marked non-blocking and no connections are present to be accepted.
[EBADF]	The descriptor is invalid.
[ECONNABORTED]	A connection has been aborted.

[EFAULT]	The *addr* parameter is not in a writable part of the user address space.
[EINTR]	The **accept**() call has been interrupted by a signal.
[EINVAL]	The socket has not been set up to accept connections (using bind(2) and listen(2)).
[EMFILE]	The per-process descriptor table is full.
[ENFILE]	The system file table is full.
[ENOTSOCK]	The descriptor references a file, not a socket.
[EOPNOTSUPP]	The referenced socket is not of type SOCK_STREAM.

SEE ALSO

bind(2), connect(2), listen(2), poll(2), select(2), socket(2)

HISTORY

The **accept**() function appeared in 4.2BSD.

NAME

access — check access permissions of a file or pathname

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
access(*const char *path*, *int mode*);

DESCRIPTION

The **access**() function checks the accessibility of the file named by *path* for the access permissions indicated by *mode*. The value of *mode* is the bitwise inclusive OR of the access permissions to be checked (R_OK for read permission, W_OK for write permission and X_OK for execute/search permission) or the existence test, F_OK. All components of the pathname *path* are checked for access permissions (including F_OK).

The real user ID is used in place of the effective user ID and the real group access list (including the real group ID) are used in place of the effective ID for verifying permission.

If a process has super-user privileges and indicates success for R_OK or W_OK, the file may not actually have read or write permission bits set. If a process has super-user privileges and indicates success for X_OK, at least one of the user, group, or other execute bits is set. (However, the file may still not be executable. See execve(2).)

RETURN VALUES

If *path* cannot be found or if any of the desired access modes would not be granted, then a −1 value is returned; otherwise a 0 value is returned.

ERRORS

Access to the file is denied if:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	The named file does not exist.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EROFS]	Write access is requested for a file on a read-only file system.
[ETXTBSY]	Write access is requested for a pure procedure (shared text) file presently being executed.
[EACCES]	Permission bits of the file mode do not permit the requested access, or search permission is denied on a component of the path prefix. The owner of a file has permission checked with respect to the ‘‘owner’’ read, write, and execute mode bits, members of the file’s group other than the owner have permission checked with respect to the ‘‘group’’ mode bits, and all others have permissions checked with respect to the ‘‘other’’ mode bits.

`[EFAULT]`	*`path`* points outside the process's allocated address space.
`[EIO]`	An I/O error occurred while reading from or writing to the file system.

SEE ALSO

`chmod`(2), `execve`(2), `stat`(2)

STANDARDS

The **`access`**() function conforms to ISO/IEC 9945-1:1990 ("POSIX.1").

BUGS

`access`() is a potential security hole and should never be used.

NAME

acct — enable or disable process accounting

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
acct(*const char *file*);

DESCRIPTION

The **acct**() call enables or disables the collection of system accounting records. If the argument *file* is a nil pointer, accounting is disabled. If *file* is an *existing* pathname (null-terminated), record collection is enabled and for every process initiated which terminates under normal conditions an accounting record is appended to *file*. Abnormal conditions of termination are reboots or other fatal system problems. Records for processes which never terminate can not be produced by **acct**().

For more information on the record structure used by **acct**(), see /usr/include/sys/acct.h and acct(5).

This call is permitted only to the super-user.

NOTES

Accounting is automatically disabled when the file system the accounting file resides on runs out of space; it is enabled when space once again becomes available. For this purpose, **acct**() creates a kernel thread called “acctwatch”.

RETURN VALUES

On error −1 is returned. The file must exist and the call may be exercised only by the super-user.

ERRORS

acct() will fail if one of the following is true:

[EPERM]	The caller is not the super-user.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix, or the path name is not a regular file.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*file* points outside the process's allocated address space.
[EIO]	An I/O error occurred while reading from or writing to the file system.

Also, **acct**() fails if failed to create kernel thread described above. See fork(2) for *errno* value.

SEE ALSO

`fork`(2), `acct`(5), `sa`(8)

HISTORY

An **`acct`**() function call appeared in Version 7 AT&T UNIX.

NAME

adjtime — correct the time to allow synchronization of the system clock

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/time.h>

int
adjtime(*const struct timeval *delta*, *struct timeval *olddelta*);

DESCRIPTION

adjtime() makes small adjustments to the system time, as returned by gettimeofday(2), advancing or retarding it by the time specified by the timeval *delta*. If *delta* is negative, the clock is slowed down by incrementing it more slowly than normal until the correction is complete. If *delta* is positive, a larger increment than normal is used. The skew used to perform the correction is generally a fraction of one percent. Thus, the time is always a monotonically increasing function. A time correction from an earlier call to **adjtime**() may not be finished when **adjtime**() is called again. If *olddelta* is non-nil, the structure pointed to will contain, upon return, the number of microseconds still to be corrected from the earlier call.

This call may be used by time servers that synchronize the clocks of computers in a local area network. Such time servers would slow down the clocks of some machines and speed up the clocks of others to bring them to the average network time.

If the calling user is not the super user, then the **adjtime**() function in the standard C library will try to use the clockctl(4) device if present, thus making possible for non privileged users to adjust the system time. If clockctl(4) is not present or not accessible, then **adjtime**() reverts to the **adjtime**() system call, which is restricted to the super user.

RETURN VALUES

A return value of 0 indicates that the call succeeded. A return value of −1 indicates that an error occurred, and in this case an error code is stored in the global variable *errno*.

ERRORS

adjtime() will fail if:

[EFAULT]	An argument points outside the process's allocated address space.
[EPERM]	The process's effective user ID is not that of the super user.

SEE ALSO

date(1), gettimeofday(2), clockctl(4), timed(8), timedc(8)

R. Gusella and S. Zatti, *TSP: The Time Synchronization Protocol for UNIX 4.3BSD*.

HISTORY

The **adjtime**() function call appeared in 4.3BSD.

NAME

arm_drain_writebuf — drains the CPU write buffer

LIBRARY

ARM Architecture Library (libarm, −larm)

SYNOPSIS

#include <machine/sysarch.h>

int
arm_drain_writebuf();

DESCRIPTION

arm_drain_writebuf() will make sure that all the entries in the processor write buffer are written out to memory.

Not all processors support this operation (currently only the SA110). Those processes that do not treat this function as a null-op.

ERRORS

arm_drain_writebuf() will never fail so will always return 0.

REFERENCES

StrongARM Data Sheet

NAME

arm_sync_icache — clean the CPU data cache and flush the CPU instruction cache

LIBRARY

ARM Architecture Library (libarm, −larm)

SYNOPSIS

#include <machine/sysarch.h>

int
arm_sync_icache(*u_int addr*, *int len*);

DESCRIPTION

arm_sync_icache() will make sure that all the entries in the processor instruction cache are synchronized with main memory and that any data in a write back cache has been cleaned. Some ARM processors (e.g. SA110) have separate instruction and data caches thus any dynamically generated or modified code needs to be written back from any data caches to main memory and the instruction cache needs to be synchronized with main memory.

On such processors **arm_sync_icache**() will clean the data cache and invalidate the processor instruction cache to force reloading from main memory. On processors that have a shared instruction and data cache and have a write through cache (e.g. ARM6) no action needs to be taken.

The routine takes a start address *addr* and a length *len* to describe the area of memory that needs to be cleaned and synchronized.

ERRORS

arm_sync_icache() will never fail so will always return 0.

REFERENCES

StrongARM Data Sheet

NAME

bind — bind a name to a socket

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/socket.h>

int
bind(*int s*, *const struct sockaddr *name*, *socklen_t namelen*);

DESCRIPTION

bind() assigns a name to an unnamed socket. When a socket is created with socket(2) it exists in a name space (address family) but has no name assigned. **bind**() requests that *name* be assigned to the socket. *namelen* indicates the amount of space pointed to by *name*, in bytes.

NOTES

Binding a name in the UNIX domain creates a socket in the file system that must be deleted by the caller when it is no longer needed (using unlink(2)).

The rules used in name binding vary between communication domains. Consult the manual entries in section 4 for detailed information.

RETURN VALUES

If the bind is successful, a 0 value is returned. A return value of −1 indicates an error, which is further specified in the global *errno*.

ERRORS

The **bind**() call will fail if:

[EBADF]	*s* is not a valid descriptor.
[ENOTSOCK]	*s* is not a socket.
[EADDRNOTAVAIL]	The specified address is not available from the local machine.
[EADDRINUSE]	The specified address is already in use.
[EINVAL]	The socket is already bound to an address.
[EINVAL]	The family of the socket and that requested in *name->sa_family* are not equivalent.
[EACCES]	The requested address is protected, and the current user has inadequate permission to access it.
[EFAULT]	The *name* parameter is not in a valid part of the user address space.

The following errors are specific to binding names in the UNIX domain.

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded NAME_MAX characters, or an entire path name exceeded PATH_MAX characters.

[ENOENT]	A prefix component of the path name does not exist.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EIO]	An I/O error occurred while making the directory entry or allocating the inode.
[EROFS]	The name would reside on a read-only file system.
[EISDIR]	An empty pathname was specified.

SEE ALSO

connect(2), getsockname(2), listen(2), socket(2)

HISTORY

The **bind**() function call appeared in 4.2BSD.

SECURITY CONSIDERATIONS

bind() was changed in NetBSD 1.4 to prevent the binding of a socket to the same port as an existing socket when all of the following is true:

- either of the existing or new addresses is INADDR_ANY,
- the uid of the new socket is not root, and the uids of the creators of the sockets are different,
- the address is not a multicast address, and
- both sockets are not bound to INADDR_ANY with SO_REUSEPORT set.

This prevents an attack where a user could bind to a port with the host's IP address (after setting SO_REUSEADDR) and 'steal' packets destined for a server that bound to the same port with INADDR_ANY.

bind() was changed in NetBSD 4.0 to honor the user's umask when binding sockets in the local domain. This was done to match the behavior of other operating systems, including FreeBSD, OpenBSD, and Linux, and to improve compatibility with some third-party software. Please note that this behavior *is not portable*. If you must bind a local socket in a portable and secure way, you need to make a directory with tight permissions and then create the socket inside it.

NAME

brk, **sbrk** — change data segment size

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
brk(*void *addr*);

*void **
sbrk(*intptr_t incr*);

DESCRIPTION

The brk and sbrk functions are legacy interfaces from before the advent of modern virtual memory management.

The **brk**() and **sbrk**() functions are used to change the amount of memory allocated in a process's data segment. They do this by moving the location of the "break". The break is the first address after the end of the process's uninitialized data segment (also known as the "BSS").

While the actual process data segment size maintained by the kernel will only grow or shrink in page sizes, these functions allow setting the break to unaligned values (i.e. it may point to any address inside the last page of the data segment).

The **brk**() function sets the break to *addr*.

The **sbrk**() function raises the break by at least *incr* bytes, thus allocating at least *incr* bytes of new memory in the data segment. If *incr* is negative, the break is lowered by *incr* bytes.

sbrk() returns the prior address of the break. The current value of the program break may be determined by calling **sbrk**(*0*). (See also end(3)).

The getrlimit(2) system call may be used to determine the maximum permissible size of the *data* segment; it will not be possible to set the break beyond the RLIMIT_DATA *rlim_max* value returned from a call to getrlimit(2), e.g. "etext + rlim.rlim_max". (see end(3) for the definition of *etext*).

RETURN VALUES

brk() returns 0 if successful; otherwise −1 with *errno* set to indicate why the allocation failed.

The **sbrk**() function returns the prior break value if successful; otherwise ((void *)−1) is returned and *errno* is set to indicate why the allocation failed.

ERRORS

brk() or **sbrk**() will fail and no additional memory will be allocated if one of the following are true:

[ENOMEM]	The limit, as set by setrlimit(2), was exceeded.
[ENOMEM]	The maximum possible size of a data segment (compiled into the system) was exceeded.
[ENOMEM]	Insufficient space existed in the swap area to support the expansion.

SEE ALSO

`execve`(2), `getrlimit`(2), `mmap`(2), `end`(3), `free`(3), `malloc`(3), `sysconf`(3)

HISTORY

A **`brk`**() function call appeared in Version 7 AT&T UNIX.

BUGS

Note that mixing **`brk`**() and **`sbrk`**() with `malloc`(3), `free`(3), and similar functions may result in non-portable program behavior. Caution is advised.

Setting the break may fail due to a temporary lack of swap space. It is not possible to distinguish this from a failure caused by exceeding the maximum size of the data segment without consulting `getrlimit`(2).

NAME

chdir, **fchdir** — change current working directory

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
chdir(*const char *path*);

int
fchdir(*int fd*);

DESCRIPTION

The *path* argument points to the pathname of a directory. The **chdir**() function causes the named directory to become the current working directory, that is, the starting point for path searches of pathnames not beginning with a slash, '/'.

The **fchdir**() function causes the directory referenced by *fd* to become the current working directory, the starting point for path searches of pathnames not beginning with a slash, '/'.

In order for a directory to become the current directory, a process must have execute (search) access to the directory.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

chdir() will fail and the current working directory will be unchanged if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	The named directory does not exist.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EACCES]	Search permission is denied for any component of the path name.
[EFAULT]	*path* points outside the process's allocated address space.
[EIO]	An I/O error occurred while reading from or writing to the file system.

fchdir() will fail and the current working directory will be unchanged if one or more of the following are true:

[EACCES]	Search permission is denied for the directory referenced by the file descriptor.
[ENOTDIR]	The file descriptor does not reference a directory.
[EBADF]	The argument *fd* is not a valid file descriptor.

`[EPERM]`	The argument *fd* references a directory which is not at or below the current process's root directory.

SEE ALSO

chroot(2)

STANDARDS

The **chdir**() function conforms to ISO/IEC 9945-1:1990 ("POSIX.1").

HISTORY

The **fchdir**() function call appeared in 4.2BSD.

NAME

chflags, **lchflags**, **fchflags** — set file flags

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/stat.h>
#include <unistd.h>

int
chflags(*const char *path*, *u_long flags*);

int
lchflags(*const char *path*, *u_long flags*);

int
fchflags(*int fd*, *u_long flags*);

DESCRIPTION

The file whose name is given by *path* or referenced by the descriptor *fd* has its flags changed to *flags*. For **lchflags**(), symbolic links are not traversed and thus their modes may be changed with this call.

The flags specified are formed by *or*'ing the following values:

UF_NODUMP	Do not dump the file.
UF_IMMUTABLE	The file may not be changed.
UF_APPEND	The file may only be appended to.
UF_OPAQUE	The file (if a directory) is opaque for union mounts.
SF_ARCHIVED	The file is archived.
SF_IMMUTABLE	The file may not be changed.
SF_APPEND	The file may only be appended to.

The UF_NODUMP, UF_IMMUTABLE, UF_APPEND, and UF_OPAQUE flags may be set or unset by either the owner of a file or the super-user, except on block and character devices, where only the super-user may set or unset them.

The SF_ARCHIVED, SF_IMMUTABLE, and SF_APPEND flags may only be set or unset by the super-user. Attempts by the non-super-user to set the super-user only flags are silently ignored. These flags may be set at any time, but normally may only be unset when the system is in single-user mode. (See init(8) for details.)

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

chflags() will fail if:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	The named file does not exist.

`[EACCES]`	Search permission is denied for a component of the path prefix.
`[ELOOP]`	Too many symbolic links were encountered in translating the pathname.
`[EPERM]`	The effective user ID does not match the owner of the file and the effective user ID is not the super-user, or the effective user ID is not the super-user and one or more of the super-user-only flags for the named file would be changed.
`[EOPNOTSUPP]`	The named file resides on a file system that does not support file flags.
`[EROFS]`	The named file resides on a read-only file system.
`[EFAULT]`	*path* points outside the process's allocated address space.
`[EIO]`	An I/O error occurred while reading from or writing to the file system.

fchflags() will fail if:

`[EBADF]`	The descriptor is not valid.
`[EINVAL]`	*fd* refers to a socket, not to a file.
`[EPERM]`	The effective user ID does not match the owner of the file and the effective user ID is not the super-user, or the effective user ID is not the super-user and one or more of the super-user-only flags for the file would be changed.
`[EOPNOTSUPP]`	The file resides on a file system that does not support file flags.
`[EROFS]`	The file resides on a read-only file system.
`[EIO]`	An I/O error occurred while reading from or writing to the file system.

SEE ALSO

chflags(1), stat(2), init(8), mount_union(8)

HISTORY

The **chflags**() and **fchflags**() functions first appeared in 4.4BSD. The **lchflags**() function first appeared in NetBSD 1.5.

NAME

chmod, **lchmod**, **fchmod** — change mode of file

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/stat.h>

int
chmod(*const char *path*, *mode_t mode*);

int
lchmod(*const char *path*, *mode_t mode*);

int
fchmod(*int fd*, *mode_t mode*);

DESCRIPTION

The function **chmod**() sets the file permission bits of the file specified by the pathname *path* to *mode*. **fchmod**() sets the permission bits of the specified file descriptor *fd*. **lchmod**() is like **chmod**() except in the case where the named file is a symbolic link, in which case **lchmod**() sets the permission bits of the link, while **chmod**() sets the bits of the file the link references. **chmod**() verifies that the process owner (user) either owns the file specified by *path* (or *fd*), or is the super-user. A mode is created from *or'd* permission bit masks defined in ⟨sys/stat.h⟩:

```
#define S_IRWXU 0000700    /* RWX mask for owner */
#define S_IRUSR 0000400    /* R for owner */
#define S_IWUSR 0000200    /* W for owner */
#define S_IXUSR 0000100    /* X for owner */

#define S_IRWXG 0000070    /* RWX mask for group */
#define S_IRGRP 0000040    /* R for group */
#define S_IWGRP 0000020    /* W for group */
#define S_IXGRP 0000010    /* X for group */

#define S_IRWXO 0000007    /* RWX mask for other */
#define S_IROTH 0000004    /* R for other */
#define S_IWOTH 0000002    /* W for other */
#define S_IXOTH 0000001    /* X for other */

#define S_ISUID 0004000    /* set user id on execution */
#define S_ISGID 0002000    /* set group id on execution */
#define S_ISVTX 0001000    /* save swapped text even after use */
```

The ISVTX (the *sticky bit*) indicates to the system which executable files are shareable (the default) and the system maintains the program text of the files in the swap area. The sticky bit may only be set by the super user on shareable executable files.

If mode ISVTX (the 'sticky bit') is set on a directory, an unprivileged user may not delete or rename files of other users in that directory. The sticky bit may be set by any user on a directory which the user owns or has appropriate permissions. For more details of the properties of the sticky bit, see sticky(7).

Changing the owner of a file turns off the set-user-id and set-group-id bits; writing to a file turns off the set-user-id and set-group-id bits unless the user is the super-user. This makes the system somewhat more secure by protecting set-user-id (set-group-id) files from remaining set-user-id (set-group-id) if they are modified, at the expense of a degree of compatibility.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

chmod() and **lchmod**() will fail and the file mode will be unchanged if:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not the super-user.
[EPERM]	The mode includes the setgid bit (S_ISGID) but the file's group is neither the effective group ID nor is it in the group access list.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*path* points outside the process's allocated address space.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[EFTYPE]	The effective user ID is not the super-user, the *mode* includes the sticky bit (S_ISVTX), and *path* does not refer to a directory.

fchmod() will fail if:

[EBADF]	The descriptor is not valid.
[EINVAL]	*fd* refers to a socket, not to a file.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not the super-user.
[EPERM]	The mode includes the setgid bit (S_ISGID) but the file's group is neither the effective group ID nor is it in the group access list.
[EROFS]	The file resides on a read-only file system.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[EFTYPE]	The effective user ID is not the super-user, the *mode* includes the sticky bit (S_ISVTX), and *fd* does not refer to a directory.

SEE ALSO

chmod(1), chflags(2), chown(2), open(2), stat(2), sticky(7), symlink(7)

STANDARDS

The **chmod**() function conforms to ISO/IEC 9945-1:1990 ("POSIX.1").

HISTORY

The **fchmod**() function call appeared in 4.2BSD. The **lchmod**() function call appeared in NetBSD 1.3.

NAME

chown, **lchown**, **fchown** — change owner and group of a file

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
chown(*const char *path*, *uid_t owner*, *gid_t group*);

int
lchown(*const char *path*, *uid_t owner*, *gid_t group*);

int
fchown(*int fd*, *uid_t owner*, *gid_t group*);

DESCRIPTION

The owner ID and group ID of the file named by *path* or referenced by *fd* is changed as specified by the arguments *owner* and *group*. The owner of a file may change the *group* to a group of which he or she is a member, but the change *owner* capability is restricted to the super-user.

When called to change the owner of a file, **chown**(), **lchown**() and **fchown**() clear the set-user-id (S_ISUID) bit on the file. When a called to change the group of a file, **chown**(), **lchown**() and **fchown**() clear the set-group-id (S_ISGID) bit on the file. These actions are taken to prevent accidental or mischievous creation of set-user-id and set-group-id programs.

lchown() is like **chown**() except in the case where the named file is a symbolic link, in which case **lchown**() changes the owner and group of the link, while **chown**() changes the owner and group of the file the link references.

fchown() is particularly useful when used in conjunction with the file locking primitives (see flock(2)).

One of the owner or group id's may be left unchanged by specifying it as (uid_t)−1 or (gid_t)−1 respectively.

RETURN VALUES

Zero is returned if the operation was successful; −1 is returned if an error occurs, with a more specific error code being placed in the global variable *errno*.

ERRORS

chown() and **lchown**() will fail and the file will be unchanged if:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EPERM]	The effective user ID is not the super-user.
[EROFS]	The named file resides on a read-only file system.

[EFAULT]	*path* points outside the process's allocated address space.
[EIO]	An I/O error occurred while reading from or writing to the file system.

fchown() will fail if:

[EBADF]	*fd* does not refer to a valid descriptor.
[EINVAL]	*fd* refers to a socket, not a file.
[EPERM]	The effective user ID is not the super-user.
[EROFS]	The named file resides on a read-only file system.
[EIO]	An I/O error occurred while reading from or writing to the file system.

SEE ALSO

chgrp(1), chmod(2), flock(2), symlink(7), chown(8)

STANDARDS

The **chown**() function deviates from the semantics defined in ISO/IEC 9945-1:1990 ("POSIX.1"), which specifies that, unless the caller is the super-user, both the set-user-id and set-group-id bits on a file shall be cleared, regardless of the file attribute changed. The **lchown**() and **fchown**() functions, as defined by X/Open Portability Guide Issue 4, Version 2 ("XPG4.2"), provide the same semantics.

To retain conformance to these standards, compatibility interfaces are provided by the POSIX Compatibility Library (libposix, −lposix) as follows:

- The **chown**() function conforms to ISO/IEC 9945-1:1990 ("POSIX.1") and X/Open Portability Guide Issue 4, Version 2 ("XPG4.2").
- The **lchown**() and **fchown**() functions conform to X/Open Portability Guide Issue 4, Version 2 ("XPG4.2").

HISTORY

The **fchown**() function call appeared in 4.2BSD.

The **chown**() and **fchown**() functions were changed to follow symbolic links in 4.4BSD. The **lchown**() function call appeared in NetBSD 1.3.

NAME

chroot — change root directory

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
chroot(*const char *dirname*);

int
fchroot(*int fd*);

DESCRIPTION

dirname is the address of the pathname of a directory, terminated by an ASCII NUL. **chroot**() causes *dirname* to become the root directory, that is, the starting point for path searches of pathnames beginning with '/'.

In order for a directory to become the root directory a process must have execute (search) access for that directory.

If the current working directory is not at or under the new root directory, it is silently set to the new root directory. It should be noted that, on most other systems, **chroot**() has no effect on the process's current directory.

This call is restricted to the super-user.

The **fchroot**() function performs the same operation on an open directory file known by the file descriptor *fd*.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate an error.

ERRORS

chroot() will fail and the root directory will be unchanged if:

[ENOTDIR]	A component of the path name is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	The named directory does not exist.
[EACCES]	Search permission is denied for any component of the path name.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EFAULT]	*dirname* points outside the process's allocated address space.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[EPERM]	The effective user ID of the calling process is not the super-user.

fchroot() will fail and the root directory will be unchanged if:

[EACCES]	Search permission is denied for the directory referenced by the file descriptor.
[EBADF]	The argument *fd* is not a valid file descriptor.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[ENOTDIR]	The argument *fd* does not reference a directory.
[EPERM]	The effective user ID of the calling process is not the super-user.

SEE ALSO

chdir(2)

STANDARDS

The **chroot**() function conforms to X/Open System Interfaces and Headers Issue 5 (“XSH5”), with the restriction that the calling process’ working directory must be at or under the new root directory. Otherwise, the working directory is silently set to the new root directory; this is an extension to the standard.

chroot() was declared a legacy interface, and subsequently removed in IEEE Std 1003.1-2001 (“POSIX.1”).

HISTORY

The **chroot**() function call appeared in 4.2BSD. Working directory handling was changed in NetBSD 1.4 to prevent one way a process could use a second **chroot**() call to a different directory to "escape" from the restricted subtree. The **fchroot**() function appeared in NetBSD 1.4.

NAME

clock_settime, **clock_gettime**, **clock_getres** — clock and timer functions

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <time.h>

int
clock_settime(*clockid_t clock_id*, *const struct timespec ∗tp*);

int
clock_gettime(*clockid_t clock_id*, *struct timespec ∗tp*);

int
clock_getres(*clockid_t clock_id*, *struct timespec ∗res*);

DESCRIPTION

The **clock_settime**() function sets the clock identified by *clock_id* to the absolute time specified by *tp*. If the time specified by *tp* is not a multiple of the resolution of the clock, *tp* is truncated to a multiple of the resolution.

clock_gettime() function stores the time of the clock identified by *clock_id* into the location specified by *tp*.

The **clock_getres**() function stores the resolution of the clock identified by *clock_id* into the location specified by *res*, unless *res* is NULL.

A *clock_id* of CLOCK_REALTIME identifies the realtime clock for the system. For this clock, the values specified by **clock_settime**() and obtained by **clock_gettime**() represent the amount of time (in seconds and nanoseconds) since 00:00 Universal Coordinated Time, January 1, 1970.

A *clock_id* of CLOCK_MONOTONIC identifies a clock that increases at a steady rate (monotonically). This clock is not affected by calls to adjtime(2) and settimeofday(2) and will fail with an EINVAL error if it's the clock specified in a call to **clock_settime**(). The origin of the clock is unspecified.

If the calling user is not the super-user, then the **clock_settime**() function in the standard C library will try to use the clockctl(4) device if present, thus making possible for non privileged users to set the system time. If clockctl(4) is not present or not accessible, then **clock_settime**() reverts to the **clock_settime**() system call, which is restricted to the super user.

RETURN VALUES

A value of 0 is returned on success. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

The **clock_settime**(), **clock_gettime**() and **clock_getres**() functions will fail if:

[EINVAL]	The *clock_id* argument does not specify a known clock.
[ENOSYS]	The function is not supported by this implementation.

The **clock_settime**() function will fail if:

[EINVAL]	The *tp* argument is outside the range for the specified clock, *clock_id*.

[EINVAL] The *tp* argument specified a nanosecond value less than zero of greater than or equal 1000 million.

[EINVAL] The *clock_id* argument is a clock that can not be adjusted.

[EPERM] The calling process does not have the appropriate privilege to set the specified clock, *clock_id*.

The **clock_gettime**() function will fail if:

[EFAULT] The *tp* argument specifies an address that is not a valid part of the process address space.

SEE ALSO

ctime(3), time(3), clockctl(4)

STANDARDS

The **clock_settime**(), **clock_gettime**() and **clock_getres**() functions conform to IEEE Std 1003.1b-1993 (“POSIX.1”).

NAME

clone — spawn new process with options

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sched.h>

pid_t
clone(*int (∗func)(void ∗arg)*, *void ∗stack*, *int flags*, *void ∗arg*);

pid_t
__clone(*int (∗func)(void ∗arg)*, *void ∗stack*, *int flags*, *void ∗arg*);

DESCRIPTION

The **clone** system call (and associated library support code) creates a new process in a way that allows the caller to specify several options for the new process creation.

Unlike fork(2) or vfork(2), in which the child process returns to the call site, **clone** causes the child process to begin execution at the function specified by *func*. The argument *arg* is passed to the entry point, as a means for the parent to provide context to the child. The stack pointer for the child process will be set to *stack*. Note that the **clone** interface requires that the application know the stack direction for the architecture, and that the caller initialize the *stack* argument as appropriate for the stack direction.

The *flags* argument specifies several options that control how the child process is created. The lower 8 bits of *flags* specify the signal that is to be sent to the parent when the child exits. The following flags may also be specified by bitwise-or'ing them with the signal value:

CLONE_VM	Share the virtual address space with the parent. The address space is shared in the same way as vfork(2).
CLONE_FS	Share the "file system information" with the parent. This include the current working directory and file creation mask.
CLONE_FILES	Share the file descriptor table with the parent.
CLONE_SIGHAND	Share the signal handler set with the parent. Note that the signal mask is never shared between the parent and the child, even if CLONE_SIGHAND is set.
CLONE_VFORK	Preserve the synchronization semantics of vfork(2); the parent blocks until the child exits.

The **clone** call returns the pid of the child in the parent's context. The child is provided no return value, since it begins execution at a different address.

If the child process's entry point returns, the value it returns is passed to _exit(2), and the child process exits. Note that if the child process wants to exit directly, it should use _exit(2), and not exit(3), since exit(3) will flush and close standard I/O channels, and thereby corrupt the parent process's standard I/O data structures (even with fork(2) it is wrong to call exit(3) since buffered data would then be flushed twice).

Note that **clone** is not intended to be used for new native NetBSD applications. It is provided as a means to port software originally written for the Linux operating system to NetBSD.

RETURN VALUES

Same as for `fork`(2).

ERRORS

Same as for `fork`(2).

SEE ALSO

`chdir`(2), `chroot`(2), `fork`(2), `sigaction`(2), `sigprocmask`(2), `umask`(2), `vfork`(2), `wait`(2)

HISTORY

The **`clone`**() function call appeared in NetBSD 1.6. It is compatible with the Linux function call of the same name.

BUGS

The NetBSD implementation of **`clone`** does not implement the `CLONE_PID` option that is present in the Linux implementation.

The NetBSD implementation of **`clone`** does not implement the `CLONE_PTRACE` option that is present in the Linux implementation.

NAME

close — delete a descriptor

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
close(*int d*);

DESCRIPTION

The **close**() system call deletes a descriptor from the per-process object reference table. If this is the last reference to the underlying object, the object will be deactivated. For example, on the last close of a file the current *seek* pointer associated with the file is lost; on the last close of a socket(2) associated naming information and queued data are discarded; on the last close of a file holding an advisory lock the lock is released (see flock(2)).

When a process exits, all associated descriptors are freed, but since there is a limit on active descriptors per processes, the **close**() system call is useful when a large quantity of file descriptors are being handled.

When a process calls fork(2), all descriptors for the new child process reference the same objects as they did in the parent before the **fork**(). If a new process is then to be run using execve(2), the process would normally inherit these descriptors. Most of the descriptors can be rearranged with dup2(2) or deleted with **close**() before the **execve**() is attempted, but if some of these descriptors will still be needed if the **execve**() fails, it is necessary to arrange for them to be closed only if the **execve**() succeeds. For this reason, the system call

fcntl(*d*, *F_SETFD*, *1*);

is provided, which arranges that a descriptor "*d*" will be closed after a successful **execve**(); the system call

fcntl(*d*, *F_SETFD*, *0*);

restores the default, which is to not close descriptor "*d*".

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

close() will fail if:

[EBADF]	*d* is not an active descriptor.
[EINTR]	An interrupt was received.

SEE ALSO

accept(2), execve(2), fcntl(2), flock(2), open(2), pipe(2), socket(2), socketpair(2)

STANDARDS

The **close**() function conforms to ISO/IEC 9945-1:1990 ("POSIX.1").

NAME

connect — initiate a connection on a socket

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/socket.h>

int
connect(*int s*, *const struct sockaddr *name*, *socklen_t namelen*);

DESCRIPTION

The parameter *s* is a socket. If it is of type SOCK_DGRAM, this call specifies the peer with which the socket is to be associated; this address is that to which datagrams are to be sent, and the only address from which datagrams are to be received. If the socket is of type SOCK_STREAM, this call attempts to make a connection to another socket. The other socket is specified by *name*, which is an address in the communications space of the socket. *namelen* indicates the amount of space pointed to by *name*, in bytes. Each communications space interprets the *name* parameter in its own way. Generally, stream sockets may successfully **connect**() only once; datagram sockets may use **connect**() multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address.

If a **connect**() call is interrupted by a signal, it will return with errno set to EINTR and the connection attempt will proceed as if the socket was non-blocking. Subsequent calls to **connect**() will set errno to EALREADY.

RETURN VALUES

If the connection or binding succeeds, 0 is returned. Otherwise a −1 is returned, and a more specific error code is stored in *errno*.

ERRORS

The **connect**() call fails if:

[EBADF]	*s* is not a valid descriptor.
[ENOTSOCK]	*s* is a descriptor for a file, not a socket.
[EADDRNOTAVAIL]	The specified address is not available on this machine.
[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this socket.
[EISCONN]	The socket is already connected.
[ETIMEDOUT]	Connection establishment timed out without establishing a connection.
[ECONNREFUSED]	The attempt to connect was forcefully rejected.
[ENETUNREACH]	The network isn't reachable from this host.
[EADDRINUSE]	The address is already in use.
[EFAULT]	The *name* parameter specifies an area outside the process address space.
[EINPROGRESS]	The socket is non-blocking and the connection cannot be completed immediately. It is possible to select(2) or poll(2) for completion by selecting or polling the socket for writing. The success or failure of the connect operation may be determined by using getsockopt(2) to read the socket error status with the SO_ERROR option at

the SOL_SOCKET level. The returned socket error status is zero on success, or one of the error codes listed here on failure.

[EALREADY]	Either the socket is non-blocking mode or a previous call to **connect**() was interrupted by a signal, and the connection attempt has not yet been completed.
[EINTR]	The connection attempt was interrupted by a signal.

The following errors are specific to connecting names in the UNIX domain. These errors may not apply in future versions of the UNIX IPC domain.

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	The named socket does not exist.
[EACCES]	Search permission is denied for a component of the path prefix, or write access to the named socket is denied.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.

SEE ALSO

accept(2), getsockname(2), getsockopt(2), poll(2), select(2), socket(2)

HISTORY

The **connect**() function call appeared in 4.2BSD.

NAME

dup, **dup2** — duplicate an existing file descriptor

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
dup(*int oldd*);

int
dup2(*int oldd*, *int newd*);

DESCRIPTION

dup() duplicates an existing object descriptor and returns its value to the calling process (*newd* = **dup**(*oldd*)). The argument *oldd* is a small non-negative integer index in the per-process descriptor table. The value must be less than the size of the table, which is returned by getdtablesize(3). The new descriptor returned by the call is the lowest numbered descriptor currently not in use by the process.

The object referenced by the descriptor does not distinguish between *oldd* and *newd* in any way. Thus if *newd* and *oldd* are duplicate references to an open file, read(2), write(2) and lseek(2) calls all move a single pointer into the file, and append mode, non-blocking I/O and asynchronous I/O options are shared between the references. If a separate pointer into the file is desired, a different object reference to the file must be obtained by issuing an additional open(2) call. The close-on-exec flag on the new file descriptor is unset.

In **dup2**(), the value of the new descriptor *newd* is specified. If this descriptor is already in use, the descriptor is first deallocated as if a close(2) call had been done first. If *newd* and *oldd* are the same, the call has no effect.

RETURN VALUES

The value −1 is returned if an error occurs in either call. The external variable *errno* indicates the cause of the error.

ERRORS

dup() and **dup2**() fail if:

[EBADF]	*oldd* or *newd* is not a valid active descriptor
[EMFILE]	Too many descriptors are active.

SEE ALSO

accept(2), close(2), fcntl(2), open(2), pipe(2), socket(2), socketpair(2), getdtablesize(3)

STANDARDS

The **dup**() and **dup2**() functions conform to ISO/IEC 9945-1:1990 (“POSIX.1”).

NAME

execve — execute a file

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
execve(*const char *path*, *char *const argv[]*, *char *const envp[]*);

DESCRIPTION

execve() transforms the calling process into a new process. The new process is constructed from an ordinary file, whose name is pointed to by *path*, called the *new process file*. This file is either an executable object file, or a file of data for an interpreter. An executable object file consists of an identifying header, followed by pages of data representing the initial program (text) and initialized data pages. Additional pages may be specified by the header to be initialized with zero data; see a.out(5).

An interpreter file begins with a line of the form:

> **#!** *interpreter* [*arg*]

When an interpreter file is **execve**()d the system runs the specified *interpreter*. If the optional *arg* is specified, it becomes the first argument to the *interpreter*, and the name of the originally **execve**()d file becomes the second argument; otherwise, the name of the originally **execve**()d file becomes the first argument. The original arguments are shifted over to become the subsequent arguments. The zeroth argument, normally the name of the **execve**()d file, is left unchanged. The interpreter named by *interpreter* must not itself be an interpreter file. (See script(7) for a detailed discussion of interpreter file execution.)

The argument *argv* is a pointer to a null-terminated array of character pointers to null-terminated character strings. These strings construct the argument list to be made available to the new process. By custom, the first element should be the name of the executed program (for example, the last component of *path*).

The argument *envp* is also a pointer to a null-terminated array of character pointers to null-terminated strings. A pointer to this array is normally stored in the global variable *environ*. These strings pass information to the new process that is not directly an argument to the command (see environ(7)).

File descriptors open in the calling process image remain open in the new process image, except for those for which the close-on-exec flag is set (see close(2) and fcntl(2)). Descriptors that remain open are unaffected by **execve**().

In the case of a new setuid or setgid executable being executed, if file descriptors 0, 1, or 2 (representing stdin, stdout, and stderr) are currently unallocated, these descriptors will be opened to point to some system file like /dev/null. The intent is to ensure these descriptors are not unallocated, since many libraries make assumptions about the use of these 3 file descriptors.

Signals set to be ignored in the calling process are set to be ignored in the new process. Signals which are set to be caught in the calling process image are set to default action in the new process image. Blocked signals remain blocked regardless of changes to the signal action. The signal stack is reset to be undefined (see sigaction(2) for more information).

If the set-user-ID mode bit of the new process image file is set (see chmod(2)), the effective user ID of the new process image is set to the owner ID of the new process image file. If the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image is set to the group ID of the new process image file. (The effective group ID is the first element of the group list.) The real user ID, real group ID and other group IDs of the new process image remain the same as the calling process image. After any set-user-ID and set-group-ID processing, the effective user ID is recorded as the saved set-user-ID, and the effective group ID is recorded as the saved set-group-ID. These values may be used in changing the effective IDs later (see `setuid`(2)).

The new process also inherits the following attributes from the calling process:

process ID	see `getpid`(2)
parent process ID	see `getppid`(2)
process group ID	see `getpgrp`(2)
access groups	see `getgroups`(2)
working directory	see `chdir`(2)
root directory	see `chroot`(2)
control terminal	see `termios`(4)
resource usages	see `getrusage`(2)
interval timers	see `getitimer`(2)
resource limits	see `getrlimit`(2)
file mode mask	see `umask`(2)
signal mask	see `sigaction`(2), `sigprocmask`(2)

When a program is executed as a result of an **execve**() call, it is entered as follows:

```
main(argc, argv, envp)
int argc;
char **argv, **envp;
```

where *argc* is the number of elements in *argv* (the “arg count”) and *argv* points to the array of character pointers to the arguments themselves.

RETURN VALUES

As the **execve**() function overlays the current process image with a new process image the successful call has no process to return to. If **execve**() does return to the calling process an error has occurred; the return value will be −1 and the global variable *errno* is set to indicate the error.

ERRORS

execve() will fail and return to the calling process if:

[EAGAIN]	A `setuid`(7) process has exceeded the current resource limit for the number of processes it is allowed to run concurrently.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	The new process file does not exist.
[ENOENT]	The new process file is a script starting with `#!` and the script interpreter does not exist.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EACCES]	Search permission is denied for a component of the path prefix, the new process file is not an ordinary file, its file mode denies execute permission, or it is on a filesystem mounted with execution disabled (`MNT_NOEXEC` in ⟨`sys/mount.h`⟩).
[ENOEXEC]	The new process file has the appropriate access permission, but has an invalid magic number in its header.

`[ETXTBSY]`	The new process file is a pure procedure (shared text) file that is currently open for writing or reading by some process.
`[ENOMEM]`	The new process requires more virtual memory than is allowed by the imposed maximum (`getrlimit`(2)).
`[E2BIG]`	The number of bytes in the new process's argument list is larger than the system-imposed limit. The limit in the system as released is 262144 bytes (`NCARGS` in ⟨`sys/param.h`⟩).
`[EFAULT]`	The new process file is not as long as indicated by the size values in its header.
`[EFAULT]`	*path*, *argv*, or *envp* point to an illegal address.
`[EIO]`	An I/O error occurred while reading from the file system.

SEE ALSO

`_exit`(2), `fork`(2), `execl`(3), `environ`(7), `script`(7)

STANDARDS

The **execve**() function conforms to ISO/IEC 9945-1:1990 ("POSIX.1").

HISTORY

The **execve**() function call first appeared in Version 7 AT&T UNIX.

BUGS

If a program is *setuid* to a non-super-user, but is executed when the real *uid* is "root", then the program has some of the powers of a super-user as well.

NAME

extattr_get_fd, **extattr_set_fd**, **extattr_delete_fd**, **extattr_list_fd**, **extattr_get_file**, **extattr_set_file**, **extattr_delete_file**, **extattr_list_file**, **extattr_get_link**, **extattr_set_link**, **extattr_delete_link**, **extattr_list_link** — system calls to manipulate VFS extended attributes

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/extattr.h>

ssize_t
extattr_get_fd(int fd, int attrnamespace, const char *attrname, void *data,
    size_t nbytes);

int
extattr_set_fd(int fd, int attrnamespace, const char *attrname,
    const void *data, size_t nbytes);

int
extattr_delete_fd(int fd, int attrnamespace, const char *attrname);

ssize_t
extattr_list_fd(int fd, int attrnamespace, void *data, size_t nbytes);

ssize_t
extattr_get_file(const char *path, int attrnamespace, const char *attrname,
    void *data, size_t nbytes);

int
extattr_set_file(const char *path, int attrnamespace, const char *attrname,
    const void *data, size_t nbytes);

int
extattr_delete_file(const char *path, int attrnamespace,
    const char *attrname);

ssize_t
extattr_list_file(const char *path, int attrnamespace, void *data,
    size_t nbytes);

ssize_t
extattr_get_link(const char *path, int attrnamespace, const char *attrname,
    void *data, size_t nbytes);

int
extattr_set_link(const char *path, int attrnamespace, const char *attrname,
    const void *data, size_t nbytes);

int
extattr_delete_link(const char *path, int attrnamespace,
    const char *attrname);

ssize_t
extattr_list_link(const char *path, int attrnamespace, void *data,
    size_t nbytes);
```

DESCRIPTION

Named extended attributes are meta-data associated with vnodes representing files and directories. They exist as "`name=value`" pairs within a set of namespaces.

The **extattr_get_file**() system call retrieves the value of the specified extended attribute into a buffer pointed to by *data* of size *nbytes*. The **extattr_set_file**() system call sets the value of the specified extended attribute to the data described by *data*. The **extattr_delete_file**() system call deletes the extended attribute specified. The **extattr_list_file**() returns a list of attributes present in the requested namespace, separated by ASCII 0 (nul) characters. The **extattr_get_file**() and **extattr_list_file**() calls consume the *data* and *nbytes* arguments in the style of `read`(2); **extattr_set_file**() consumes these arguments in the style of `write`(2).

If *data* is `NULL` in a call to **extattr_get_file**() then the size of defined extended attribute data will be returned, rather than the quantity read, permitting applications to test the size of the data without performing a read.

The **extattr_delete_link**(), **extattr_get_link**(), and **extattr_set_link**() system calls behave in the same way as their _file counterparts, except that they do not follow symlinks.

The **extattr_get_fd**(), **extattr_set_fd**(), and **extattr_delete_fd**() calls are identical to their "`_file`" counterparts except for the first argument. The "`_fd`" functions take a file descriptor, while the "`_file`" functions take a path. Both arguments describe a file associated with the extended attribute that should be manipulated.

The following arguments are common to all the system calls described here:

attrnamespace	the namespace in which the extended attribute resides; see `extattr`(9)
attrname	the name of the extended attribute

Named extended attribute semantics vary by file system implementing the call. Not all operations may be supported for a particular attribute. Additionally, the format of the data in *data* is attribute-specific.

For more information on named extended attributes, please see `extattr`(9).

RETURN VALUES

If successful, the **extattr_get_file**() and **extattr_set_file**() calls return the number of bytes that were read or written from the *data*, respectively, or if *data* was `NULL`, then **extattr_get_file**() returns the number of bytes available to read. If any of the calls are unsuccessful, the value −1 is returned and the global variable *errno* is set to indicate the error.

The **extattr_delete_file**() function returns the value 0 if successful; otherwise the value −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The following errors may be returned by the system calls themselves. Additionally, the file system implementing the call may return any other errors it desires.

[EFAULT]	The *attrnamespace* and *attrname* arguments, or the memory range defined by *data* and *nbytes* point outside the process's allocated address space.
[ENAMETOOLONG]	The attribute name was longer than `EXTATTR_MAXNAMELEN`.

The **extattr_get_fd**(), **extattr_set_fd**(), and **extattr_delete_fd**() system calls may also fail if:

[EBADF]	The file descriptor referenced by *fd* was invalid.

Additionally, the **extattr_get_file**(), **extattr_set_file**(), and **extattr_delete_file**() calls may also fail due to the following errors:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.
[ENOENT]	A component of the path name that must exist does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.

SEE ALSO

getextattr(1), extattr(3), extattr(9)

HISTORY

Extended attribute support was developed as part of the TrustedBSD Project, and introduced in FreeBSD 5.0 and NetBSD 3.0. It was developed to support security extensions requiring additional labels to be associated with each file or directory.

CAVEATS

This interface is under active development, and as such is subject to change as applications are adapted to use it. Developers are discouraged from relying on its stability.

NAME

fcntl — file descriptor control

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <fcntl.h>

int
fcntl(*int fd*, *int cmd*, *...*);

DESCRIPTION

fcntl() provides for control over descriptors. The argument *fd* is a descriptor to be operated on by *cmd* as described below. The third parameter is called *arg* and is technically a pointer to void, but it is interpreted as an int by some commands and ignored by others.

Commands are:

F_DUPFD	Return a new descriptor as follows: • Lowest numbered available descriptor greater than or equal to *arg*, which is interpreted as an int. • Same object references as the original descriptor. • New descriptor shares the same file offset if the object was a file. • Same access mode (read, write or read/write). • Same file status flags (i.e., both file descriptors share the same file status flags). • The close-on-exec flag associated with the new file descriptor is cleared to remain open across execve(2) system calls.
F_GETFD	Get the close-on-exec flag associated with the file descriptor *fd* as FD_CLOEXEC. If the returned value ANDed with FD_CLOEXEC is 0, the file will remain open across **exec**(), otherwise the file will be closed upon execution of **exec**() (*arg* is ignored).
F_SETFD	Set the close-on-exec flag associated with *fd* to *arg*, where *arg* is either 0 or FD_CLOEXEC, as described above.
F_GETFL	Get descriptor status flags, as described below (*arg* is ignored).
F_SETFL	Set descriptor status flags to *arg*, which is interpreted as an int.
F_GETOWN	Get the process ID or process group currently receiving SIGIO and SIGURG signals; process groups are returned as negative values (*arg* is ignored).
F_SETOWN	Set the process or process group to receive SIGIO and SIGURG signals; process groups are specified by supplying *arg* as negative, otherwise *arg* is interpreted as a process ID. The argument *arg* is interpreted as an int.
F_CLOSEM	Close all file descriptors greater than or equal to *fd*.
F_MAXFD	Return the maximum file descriptor number currently open by the process.

The flags for the F_GETFL and F_SETFL flags are as follows:

O_NONBLOCK	Non-blocking I/O; if no data is available to a read(2) call, or if a write(2) operation would block, the read or write call returns −1 with the error EAGAIN.

`O_APPEND` Force each write to append at the end of file; corresponds to the `O_APPEND` flag of `open`(2).

`O_ASYNC` Enable the `SIGIO` signal to be sent to the process group when I/O is possible, e.g., upon availability of data to be read.

Several commands are available for doing advisory file locking; they all operate on the following structure:

```
struct flock {
        off_t   l_start;        /* starting offset */
        off_t   l_len;          /* len = 0 means until end of file */
        pid_t   l_pid;          /* lock owner */
        short   l_type;         /* lock type: read/write, etc. */
        short   l_whence;       /* type of l_start */
};
```

The commands available for advisory record locking are as follows:

`F_GETLK` Get the first lock that blocks the lock description pointed to by the third argument, *arg*, taken as a pointer to a *struct flock* (see above). The information retrieved overwrites the information passed to **fcntl** in the *flock* structure. If no lock is found that would prevent this lock from being created, the structure is left unchanged by this function call except for the lock type *l_type*, which is set to `F_UNLCK`.

`F_SETLK` Set or clear a file segment lock according to the lock description pointed to by the third argument, *arg*, taken as a pointer to a *struct flock* (see above). As specified by the value of *l_type*, `F_SETLK` is used to establish shared (or read) locks (`F_RDLCK`) or exclusive (or write) locks, (`F_WRLCK`), as well as remove either type of lock (`F_UNLCK`). If a shared or exclusive lock cannot be set, **fcntl** returns immediately with `EAGAIN`.

`F_SETLKW` This command is the same as `F_SETLK` except that if a shared or exclusive lock is blocked by other locks, the process waits until the request can be satisfied. If a signal that is to be caught is received while **fcntl** is waiting for a region, the **fcntl** will be interrupted if the signal handler has not specified the `SA_RESTART` (see `sigaction`(2)).

When a shared lock has been set on a segment of a file, other processes can set shared locks on that segment or a portion of it. A shared lock prevents any other process from setting an exclusive lock on any portion of the protected area. A request for a shared lock fails if the file descriptor was not opened with read access.

An exclusive lock prevents any other process from setting a shared lock or an exclusive lock on any portion of the protected area. A request for an exclusive lock fails if the file was not opened with write access.

The value of *l_whence* is `SEEK_SET`, `SEEK_CUR`, or `SEEK_END` to indicate that the relative offset, *l_start* bytes, will be measured from the start of the file, current position, or end of the file, respectively. The value of *l_len* is the number of consecutive bytes to be locked. If *l_len* is negative, the result is undefined. The *l_pid* field is only used with `F_GETLK` to return the process ID of the process holding a blocking lock. After a successful `F_GETLK` request, the value of *l_whence* is `SEEK_SET`.

Locks may start and extend beyond the current end of a file, but may not start or extend before the beginning of the file. A lock is set to extend to the largest possible value of the file offset for that file if *l_len* is set to zero. If *l_whence* and *l_start* point to the beginning of the file, and *l_len* is zero, the entire file is locked. If an application wishes only to do entire file locking, the `flock`(2) system call is much more efficient.

There is at most one type of lock set for each byte in the file. Before a successful return from an `F_SETLK` or an `F_SETLKW` request when the calling process has previously existing locks on bytes in the region specified by the request, the previous lock type for each byte in the specified region is replaced by the new lock type. As specified above under the descriptions of shared locks and exclusive locks, an `F_SETLK` or an

`F_SETLKW` request fails or blocks respectively when another process has existing locks on bytes in the specified region and the type of any of those locks conflicts with the type specified in the request.

This interface follows the completely stupid semantics of AT&T System V UNIX and IEEE Std 1003.1-1988 ("POSIX.1") that require that all locks associated with a file for a given process are removed when *any* file descriptor for that file is closed by that process. This semantic means that applications must be aware of any files that a subroutine library may access. For example if an application for updating the password file locks the password file database while making the update, and then calls `getpwnam`(3) to retrieve a record, the lock will be lost because `getpwnam`(3) opens, reads, and closes the password database. The database close will release all locks that the process has associated with the database, even if the library routine never requested a lock on the database. Another minor semantic problem with this interface is that locks are not inherited by a child process created using the `fork`(2) function. The `flock`(2) interface has much more rational last close semantics and allows locks to be inherited by child processes. Calling `flock`(2) is recommended for applications that want to ensure the integrity of their locks when using library routines or wish to pass locks to their children. Note that `flock`(2) and **`fcntl`** locks may be safely used concurrently.

All locks associated with a file for a given process are removed when the process terminates.

A potential for deadlock occurs if a process controlling a locked region is put to sleep by attempting to lock the locked region of another process. This implementation detects that sleeping until a locked region is unlocked would cause a deadlock and fails with an `EDEADLK` error.

RETURN VALUES

Upon successful completion, the value returned depends on *`cmd`* as follows:

`F_DUPFD`	A new file descriptor.
`F_GETFD`	Value of flag (only the low-order bit is defined).
`F_GETFL`	Value of flags.
`F_GETOWN`	Value of file descriptor owner.
`F_MAXFD`	Value of the highest file descriptor open by the process.
other	Value other than −1.

Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

`fcntl`() will fail if:

[EAGAIN]	The argument *`arg`* is `F_SETLK`, the type of lock (*`l_type`*) is a shared lock (`F_RDLCK`) or exclusive lock (`F_WRLCK`), and the segment of a file to be locked is already exclusive-locked by another process; or the type is an exclusive lock and some portion of the segment of a file to be locked is already shared-locked or exclusive-locked by another process.
[EBADF]	*`fildes`* is not a valid open file descriptor.
	The argument *`cmd`* is `F_SETLK` or `F_SETLKW`, the type of lock (*`l_type`*) is a shared lock (`F_RDLCK`), and *`fildes`* is not a valid file descriptor open for reading.
	The argument *`cmd`* is `F_SETLK` or `F_SETLKW`, the type of lock (*`l_type`*) is an exclusive lock (`F_WRLCK`), and *`fildes`* is not a valid file descriptor open for writing.

`[EDEADLK]`	The argument *cmd* is `F_SETLKW`, and a deadlock condition was detected.
`[EINTR]`	The argument *cmd* is `F_SETLKW`, and the function was interrupted by a signal.
`[EINVAL]`	*cmd* is `F_DUPFD` and *arg* is negative or greater than the maximum allowable number (see `getdtablesize`(3)). The argument *cmd* is `F_GETLK`, `F_SETLK`, or `F_SETLKW` and the data to which *arg* points is not valid, or *fildes* refers to a file that does not support locking.
`[EMFILE]`	The argument *cmd* is `F_DUPFD` and the maximum number of file descriptors permitted for the process are already in use, or no file descriptors greater than or equal to *arg* are available.
`[ENFILE]`	*cmd* is `F_DUPFD` and system-wide the maximum allowed number of file descriptors are currently open.
`[ENOLCK]`	The argument *cmd* is `F_SETLK` or `F_SETLKW`, and satisfying the lock or unlock request would result in the number of locked regions in the system exceeding a system-imposed limit.
`[ESRCH]`	*cmd* is `F_SETOWN` and the process ID given as argument is not in use.

SEE ALSO

`close`(2), `execve`(2), `flock`(2), `open`(2), `sigaction`(2), `getdtablesize`(3)

STANDARDS

The **`fcntl`**() function conforms to ISO/IEC 9945-1:1990 ("POSIX.1").

HISTORY

The **`fcntl`**() function call appeared in 4.2BSD.

NAME

fdatasync — synchronize the data of a file

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
fdatasync(*int fd*);

DESCRIPTION

The **fdatasync**() function forces all modified data associated with the file descriptor *fd* to be flushed to stable storage.

The functionality is as described for fsync(2), with the exception that file status information need not be synchronized, which may result in a performance gain, compared to fsync(2). This behaviour is commonly known as **synchronized I/O data integrity completion.**

RETURN VALUES

A value of 0 is returned on success. Otherwise, a value −1 is returned and *errno* is set to indicate the error.

ERRORS

The **fdatasync**() function will fail if:

[EBADF]	The *fd* argument is not a valid file descriptor open for writing.
[EINVAL]	This implementation does not support synchronized I/O for this file.
[ENOSYS]	The **fdatasync**() function is not supported by this implementation.

In the event that any of the I/O operations to be performed fail, **fdatasync**() returns the error conditions defined for read(2) and write(2), and outstanding I/O operations are not guaranteed to have been completed.

SEE ALSO

fsync(2), open(2), read(2), write(2)

STANDARDS

The **fdatasync**() function conforms to IEEE Std 1003.1b-1993 (“POSIX.1”).

NAME

fhopen, **fhstat**, **fhstatvfs** — access file via file handle

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/types.h>

int
fhopen(*const void ∗fhp*, *size_t fh_size*, *int flags*);

#include <sys/stat.h>

int
fhstat(*const void ∗fhp*, *size_t fh_size*, *struct stat ∗sb*);

#include <sys/statvfs.h>

int
fhstatvfs(*const void ∗fhp*, *size_t fh_size*, *struct statvfs ∗buf*);

int
fhstatvfs1(*const void ∗fhp*, *size_t fh_size*, *struct statvfs ∗buf*, *int flags*);

DESCRIPTION

These functions provide a means to access a file given the opaque file handle *fhp* and the size *fh_size* of the opaque object as returned by getfh(2). As this method bypasses directory access restrictions, these calls are restricted to the superuser.

fhopen() opens the file referenced by *fhp* for reading and/or writing as specified by the argument *flags* and returns the file descriptor to the calling process. The *flags* are specified by *or*'ing together the flags used for the open(2) call. All said flags are valid except for O_CREAT.

fhstat(), **fhstatvfs**(), and **fhstatvfs1**() provide the functionality of the fstat(2), fstatvfs(2), and fstatvfs1(2) calls except that they return information for the file referred to by *fhp* rather than an open file.

RETURN VALUES

Upon successful completion, **fhopen**() returns the file descriptor for the opened file, while **fhstat**(), **fhstatvfs**(), and **fhstatvfs1**() return 0. Otherwise, −1 is returned and *errno* is set to indicate the error.

ERRORS

In addition to the errors returned by open(2), fstat(2), fstatvfs(2), and fstatvfs1(2), respectively, **fhopen**(), **fhstat**(), **fhstatvfs**(), and **fhstatvfs1**() will return

[EINVAL]	Calling **fhopen**() with O_CREAT set or invalid *fh_size*.
[ESTALE]	The file handle *fhp* is no longer valid.

SEE ALSO

fstat(2), fstatvfs(2), fstatvfs1(2), getfh(2), open(2)

HISTORY

The **fhopen**(), and **fhstat**() functions first appeared in NetBSD 1.5. The **fhstatvfs**() function replaced **fhstatfs**() in NetBSD 3.0.

NAME

flock — apply or remove an advisory lock on an open file

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <fcntl.h>
#define     LOCK_SH     1     /* shared lock */
#define     LOCK_EX     2     /* exclusive lock */
#define     LOCK_NB     4     /* don't block when locking */
#define     LOCK_UN     8     /* unlock */
```

int
flock(*int fd*, *int operation*);

DESCRIPTION

flock() applies or removes an *advisory* lock on the file associated with the file descriptor *fd*. A lock is applied by specifying an *operation* parameter that is one of LOCK_SH or LOCK_EX with the optional addition of LOCK_NB. To unlock an existing lock operation should be LOCK_UN.

Advisory locks allow cooperating processes to perform consistent operations on files, but do not guarantee consistency (i.e., processes may still access files without using advisory locks possibly resulting in inconsistencies).

The locking mechanism allows two types of locks: *shared* locks and *exclusive* locks. At any time multiple shared locks may be applied to a file, but at no time are multiple exclusive, or both shared and exclusive, locks allowed simultaneously on a file.

A shared lock may be *upgraded* to an exclusive lock, and vice versa, simply by specifying the appropriate lock type; this results in the previous lock being released and the new lock applied (possibly after other processes have gained and released the lock).

Requesting a lock on an object that is already locked normally causes the caller to be blocked until the lock may be acquired. If LOCK_NB is included in *operation*, then this will not happen; instead the call will fail and the error EAGAIN will be returned.

NOTES

Locks are on files, not file descriptors. That is, file descriptors duplicated through dup(2) or fork(2) do not result in multiple instances of a lock, but rather multiple references to a single lock. If a process holding a lock on a file forks and the child explicitly unlocks the file, the parent will lose its lock.

Processes blocked awaiting a lock may be awakened by signals.

RETURN VALUES

Zero is returned if the operation was successful; on an error a −1 is returned and an error code is left in the global location *errno*.

ERRORS

The **flock**() call fails if:

[EAGAIN]	The file is locked and the LOCK_NB option was specified.
[EBADF]	The argument *fd* is an invalid descriptor.

[EOPNOTSUPP]	The argument *fd* refers to an object other than a file.
[EINVAL]	The argument *operation* does not include one of LOCK_EX, LOCK_SH or LOCK_UN.

SEE ALSO

close(2), dup(2), execve(2), fork(2), open(2)

HISTORY

The **flock**() function call appeared in 4.2BSD.

NAME

fork — create a new process

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

pid_t
fork(*void*);

DESCRIPTION

fork() causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process) except for the following:

- The child process has a unique process ID.
- The child process has a different parent process ID (i.e., the process ID of the parent process).
- The child process has its own copy of the parent's descriptors. These descriptors reference the same underlying objects, so that, for instance, file pointers in file objects are shared between the child and the parent, so that an `lseek`(2) on a descriptor in the child process can affect a subsequent `read`(2) or `write`(2) by the parent. This descriptor copying is also used by the shell to establish standard input and output for newly created processes as well as to set up pipes.
- The child process' resource utilizations are set to 0; see `setrlimit`(2).

In general, the child process should call `_exit`(2) rather than `exit`(3). Otherwise, any stdio buffers that exist both in the parent and child will be flushed twice. Similarly, `_exit`(2) should be used to prevent `atexit`(3) routines from being called twice (once in the parent and once in the child).

In case of a threaded program, only the thread calling **fork**() is still running in the child processes.

Child processes of a threaded program have additional restrictions, a child must only call functions that are async-signal-safe. Very few functions are asynchronously safe and applications should make sure they call `exec`(3) as soon as possible.

RETURN VALUES

Upon successful completion, **fork**() returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of −1 is returned to the parent process, no child process is created, and the global variable *errno* is set to indicate the error.

ERRORS

fork() will fail and no child process will be created if:

[EAGAIN]	The system-imposed limit on the total number of processes under execution would be exceeded. This limit is configuration-dependent.
[EAGAIN]	The limit RLIMIT_NPROC on the total number of processes under execution by this user id would be exceeded.
[ENOMEM]	There is insufficient swap space for the new process.

SEE ALSO

`execve`(2), `setrlimit`(2), `vfork`(2), `wait`(2), `pthread_atfork`(3)

STANDARDS

The **`fork`**() function conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

HISTORY

A **`fork`**() system call appeared in Version 6 AT&T UNIX.

NAME

fsync, **fsync_range** — synchronize a file's in-core state with that on disk

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
fsync(*int fd*);

int
fsync_range(*int fd*, *int how*, *off_t start*, *off_t length*);

DESCRIPTION

fsync() causes all modified data and attributes of *fd* to be moved to a permanent storage device. This normally results in all in-core modified copies of buffers for the associated file to be written to a disk.

fsync() should be used by programs that require a file to be in a known state, for example, in building a simple transaction facility.

fsync_range() causes all modified data starting at *start* for length *length* of *fd* to be written to permanent storage. Note that **fsync_range**() requires that the file *fd* must be open for writing.

fsync_range() may flush the file data in one of two manners:

FDATASYNC	Synchronize the file data and sufficient meta-data to retrieve the data for the specified range.
FFILESYNC	Synchronize all modified file data and meta-data for the specified range.

By default, **fsync_range**() does not flush disk caches, assuming that storage media are able to ensure completed writes are transfered to media. The *FDISKSYNC* flag may be included in the *how* parameter to trigger flushing of all disk caches for the file.

If the *length* parameter is zero, **fsync_range**() will synchronize all of the file data.

RETURN VALUES

A 0 value is returned on success. A −1 value indicates an error.

ERRORS

fsync() or **fsync_range**() fail if:

[EBADF]	*fd* is not a valid descriptor.
[EINVAL]	*fd* refers to a socket, not to a file.
[EIO]	An I/O error occurred while reading from or writing to the file system.

Additionally, **fsync_range**() fails if:

[EBADF]	*fd* is not open for writing.
[EINVAL]	*start* + *length* is less than *start*.

NOTES

For optimal efficiency, the **fsync_range**() call requires that the file system containing the file referenced by *fd* support partial synchronization of file data. For file systems which do not support partial synchronization, the entire file will be synchronized and the call will be the equivalent of calling **fsync**().

SEE ALSO

sync(2), sync(8)

HISTORY

The **fsync**() function call appeared in 4.2BSD.

The **fsync_range**() function call first appeared in NetBSD 2.0 and is modeled after the function available in AIX.

NAME

getcontext, **setcontext** — get and set current user context

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <ucontext.h>

int
getcontext(*ucontext_t *ucp*);

int
setcontext(*const ucontext_t *ucp*);

DESCRIPTION

The **getcontext**() function initializes the object pointed to by *ucp* to the current user context of the calling thread. The user context defines a thread's execution environment and includes the contents of its machine registers, its signal mask, and its current execution stack.

The **setcontext**() function restores the user context defined in the object pointed to by *ucp* as most recently initialized by a previous call to either **getcontext**() or makecontext(3). If successful, execution of the program resumes as defined in the *ucp* argument, and **setcontext**() will not return. If *ucp* was initialized by the **getcontext**() function, program execution continues as if the corresponding invocation of **getcontext**() had just returned (successfully). If *ucp* was initialized by the makecontext(3) function, program execution continues with the function (and function arguments) passed to makecontext(3).

RETURN VALUES

On successful completion, **getcontext**() returns 0 and **setcontext**() does not return. Otherwise a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

The **getcontext**() and **setcontext**() functions will fail if:

[EFAULT]	The *ucp* argument points to an invalid address.

The **setcontext**() function will fail if:

[EINVAL]	The contents of the datum pointed to by *ucp* are invalid.

SEE ALSO

sigprocmask(2), longjmp(3), makecontext(3), setjmp(3), swapcontext(3)

STANDARDS

The **getcontext**() and **setcontext**() functions conform to X/Open System Interfaces and Headers Issue 5 ("XSH5"). The *errno* indications are an extension to the standard.

HISTORY

The **getcontext**() and **setcontext**() functions first appeared in AT&T System V.4 UNIX.

NAME

getdents — get directory entries in a filesystem independent format

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <dirent.h>

int
getdents(*int fd*, *char *buf*, *size_t nbytes*);

DESCRIPTION

getdents() reads directory entries from the directory referenced by the file descriptor *fd* into the buffer pointed to by *buf*, in a filesystem independent format. Up to *nbytes* of data will be transferred. *nbytes* must be greater than or equal to the block size associated with the file, see stat(2). Some filesystems may not support **getdents**() with buffers smaller than this size.

The data in the buffer is a series of *dirent* structures each containing the following entries:

```
ino_t           d_fileno;
uint16_t        d_reclen;
uint16_t        d_namlen;
uint8_t d_type;
char            d_name[MAXNAMLEN + 1]; /* see below */
```

The *d_fileno* entry is a number which is unique for each distinct file in the filesystem. Files that are linked by hard links (see link(2)) have the same *d_fileno*. If *d_fileno* is zero, the entry refers to a deleted file.

The *d_reclen* entry is the length, in bytes, of the directory record.

The *d_type* is the type of file, where the following are possible types: DT_UNKNOWN, DT_FIFO, DT_CHR, DT_DIR, DT_BLK, DT_REG, DT_LNK, DT_SOCK, and DT_WHT.

The *d_namlen* entry specifies the length of the file name excluding the null byte. Thus the actual size of *d_name* may vary from 1 to MAXNAMLEN + 1.

The *d_name* entry contains a null terminated file name.

Entries may be separated by extra space. The *d_reclen* entry may be used as an offset from the start of a *dirent* structure to the next structure, if any.

The actual number of bytes transferred is returned. The current position pointer associated with *fd* is set to point to the next block of entries. The pointer may not advance by the number of bytes returned by **getdents**(). A value of zero is returned when the end of the directory has been reached.

The current position pointer may be set and retrieved by lseek(2). The current position pointer should only be set to a value returned by lseek(2), or zero.

RETURN VALUES

If successful, the number of bytes actually transferred is returned. Otherwise, −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

getdents() will fail if:

[EBADF]	*fd* is not a valid file descriptor open for reading.
[EFAULT]	Either *buf* points outside the allocated address space.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[EINVAL]	A directory was being read on NFS, but it was modified on the server while it was being read.

SEE ALSO

lseek(2), open(2), dirent(5)

HISTORY

The **getdents**() function first appeared in NetBSD 1.3.

NAME

getfh — get file handle

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/mount.h>

int
getfh(*const char *path*, *void *fhp*, *size_t *fh_size*);

DESCRIPTION

getfh() returns a file handle for the specified file or directory in the file handle pointed to by *fhp*. The variable pointed to by *fh_size* has to be initialized to the memory allocated for the variable sized file handle. On return the value will be replaced by the actual size needed (which will vary depending on the file system the path is on). This system call is restricted to the superuser. To query the necessary size for the filehandle, a NULL pointer may be passed as *fhp*, and the value pointed to by *fh_size* should be initialized to zero.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

getfh() fails if one or more of the following are true:

[ENOTDIR]	A component of the path prefix of *path* is not a directory.
[ENAMETOOLONG]	The length of a component of *path* exceeds {NAME_MAX} characters, or the length of *path* exceeds {PATH_MAX} characters.
[ENOENT]	The file referred to by *path* does not exist.
[EACCES]	Search permission is denied for a component of the path prefix of *path*.
[ELOOP]	Too many symbolic links were encountered in translating *path*.
[EFAULT]	*fhp* points to an invalid address.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[E2BIG]	The memory allocated for the file handle is too small. The size needed has been written to the variable pointed to by *fh_size*.
[ENOMEM]	The kernel failed to allocate temporary memory to create a filehandle of the requested size.

SEE ALSO

fhstat(2), fhstatvfs(2), fhstatvfs1(2)

HISTORY

The **getfh**() function first appeared in 4.4BSD.

NAME

getgid, **getegid** — get group process identification

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

gid_t
getgid(*void*);

gid_t
getegid(*void*);

DESCRIPTION

The **getgid**() function returns the real group ID of the calling process, **getegid**() returns the effective group ID of the calling process.

The real group ID is specified at login time.

The real group ID is the group of the user who invoked the program. As the effective group ID gives the process additional permissions during the execution of "*set-group-ID*" mode processes, **getgid**() is used to determine the real-group-id of the calling process.

ERRORS

The **getgid**() and **getegid**() functions are always successful, and no return value is reserved to indicate an error.

SEE ALSO

getuid(2), setgid(2), setgroups(2), setregid(2)

STANDARDS

getgid() and **getegid**() conform to ISO/IEC 9945-1:1990 ("POSIX.1").

NAME

getgroups — get group access list

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
getgroups(*int gidsetlen*, *gid_t *gidset*);

DESCRIPTION

getgroups() gets the current group access list of the current user process and stores it in the array *gidset*. The parameter *gidsetlen* indicates the number of entries that may be placed in *gidset*. **getgroups**() returns the actual number of groups returned in *gidset*. No more than {NGROUPS_MAX} will ever be returned. If *gidsetlen* is 0, **getgroups**() returns the number of groups without modifying the *gidset* array.

This system call only returns the secondary groups.

RETURN VALUES

A successful call returns the number of groups in the group set. A value of −1 indicates that an error occurred, and the error code is stored in the global variable *errno*.

ERRORS

The possible errors for **getgroups**() are:

[EINVAL]	The argument *gidsetlen* is non-zero and is smaller than the number of groups in the group set.
[EFAULT]	The argument *gidset* specifies an invalid address.

SEE ALSO

getegid(2), getgid(2), setgroups(2), initgroups(3)

STANDARDS

The **getgroups**() function conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

HISTORY

The **getgroups**() function call appeared in 4.2BSD.

NAME

getitimer, **setitimer** — get/set value of interval timer

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/time.h>
#define ITIMER_REAL         0
#define ITIMER_VIRTUAL  1
#define ITIMER_PROF         2
```

int
getitimer(*int which*, *struct itimerval ∗value*);

int
setitimer(*int which*, *const struct itimerval ∗ restrict value*,
    struct itimerval ∗ restrict ovalue);

DESCRIPTION

The system provides each process with three interval timers, defined in ⟨sys/time.h⟩. The **getitimer**() call returns the current value for the timer specified in *which* in the structure at *value*. The **setitimer**() call sets a timer to the specified *value* (returning the previous value of the timer if *ovalue* is non-nil).

A timer value is defined by the *itimerval* structure:

```
struct itimerval {
        struct  timeval it_interval;   /∗ timer interval ∗/
        struct  timeval it_value;      /∗ current value ∗/
};
```

If *it_value* is non-zero, it indicates the time to the next timer expiration. If *it_interval* is non-zero, it specifies a value to be used in reloading *it_value* when the timer expires. Setting *it_value* to 0 disables a timer. Setting *it_interval* to 0 causes a timer to be disabled after its next expiration (assuming *it_value* is non-zero).

Time values smaller than the resolution of the system clock are rounded up to this resolution (typically 10 milliseconds).

The ITIMER_REAL timer decrements in real time. A SIGALRM signal is delivered when this timer expires.

The ITIMER_VIRTUAL timer decrements in process virtual time. It runs only when the process is executing. A SIGVTALRM signal is delivered when it expires.

The ITIMER_PROF timer decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by interpreters in statistically profiling the execution of interpreted programs. Each time the ITIMER_PROF timer expires, the SIGPROF signal is delivered. Because this signal may interrupt in-progress system calls, programs using this timer must be prepared to restart interrupted system calls.

NOTES

Macros for manipulating time values are defined in ⟨sys/time.h⟩. **timerclear**() sets a time value to zero, **timerisset**() tests if a time value is non-zero, **timercmp**() compares two time values, **timeradd**() adds a time value to another time value, **timersub**() computes the time difference between two time values.

RETURN VALUES

If the calls succeed, a value of 0 is returned. If an error occurs, the value −1 is returned, and a more precise error code is placed in the global variable *errno*.

ERRORS

getitimer() and **setitimer**() will fail if:

[EFAULT]	The *value* parameter specified a bad address.
[EINVAL]	A *value* parameter specified a time that was too large to be handled.

SEE ALSO

gettimeofday(2), poll(2), select(2), sigaction(2)

HISTORY

The **getitimer**() function call appeared in 4.2BSD.

NAME

getlogin, **setlogin** — get/set login name

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

char ∗
getlogin(*void*);

int
setlogin(*const char ∗name*);

DESCRIPTION

The **getlogin**() routine returns the login name of the user associated with the current session, as previously set by **setlogin**(). The name is normally associated with a login shell at the time a session is created, and is inherited by all processes descended from the login shell. (This is true even if some of those processes assume another user ID, for example when su(1) is used.)

setlogin() sets the login name of the user associated with the current session to *name*. This call is restricted to the super-user, and is normally used only when a new session is being created on behalf of the named user (for example, at login time, or when a remote shell is invoked).

NOTE: There is only one login name per session.

It is *CRITICALLY* important to ensure that **setlogin**() is only ever called after the process has taken adequate steps to ensure that it is detached from its parent's session. The *ONLY* way to do this is via the **setsid**() function. The **daemon**() function calls **setsid**() which is an ideal way of detaching from a controlling terminal and forking into the background.

In particular, neither **ioctl**(*ttyfd*, *TIOCNOTTY*, *. . .*) nor **setpgid**(*. . .*) is sufficient to create a new session.

Once a parent process has called **setsid**(), it is acceptable for some child of that process to then call **setlogin**(), even though it is not the session leader. Beware, however, that *ALL* processes in the session will change their login name at the same time, even the parent.

This is different from traditional UNIX privilege inheritance and as such can be counter-intuitive.

Since the **setlogin**() routine is restricted to the super-user, it is assumed that (like all other privileged programs) the programmer has taken adequate precautions to prevent security violations.

RETURN VALUES

If a call to **getlogin**() succeeds, it returns a pointer to a null-terminated string in a static buffer. If the name has not been set, it returns NULL. If a call to **setlogin**() succeeds, a value of 0 is returned. If **setlogin**() fails, a value of −1 is returned and an error code is placed in the global location *errno*.

ERRORS

The following errors may be returned by these calls:

[EFAULT]	The *name* parameter gave an invalid address.
[EINVAL]	The *name* parameter pointed to a string that was too long. Login names are limited to MAXLOGNAME (from ⟨sys/param.h⟩) characters, currently 16.

[EPERM] The caller tried to set the login name and was not the super-user.

SEE ALSO

setsid(2)

STANDARDS

The **getlogin**() function conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

HISTORY

The **getlogin**() function first appeared in 4.4BSD.

BUGS

Login names are limited in length by **setlogin**(). However, lower limits are placed on login names elsewhere in the system (UT_NAMESIZE in ⟨utmp.h⟩).

In earlier versions of the system, **getlogin**() failed unless the process was associated with a login terminal. The current implementation (using **setlogin**()) allows getlogin to succeed even when the process has no controlling terminal. In earlier versions of the system, the value returned by **getlogin**() could not be trusted without checking the user ID. Portable programs should probably still make this check.

NAME

getpeername — get name of connected peer

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/socket.h>

int
getpeername(*int s*, *struct sockaddr * restrict name*,
    *socklen_t * restrict namelen*);

DESCRIPTION

getpeername() returns the name of the peer connected to socket *s*. One common use occurs when a process inherits an open socket, such as TCP servers forked from inetd(8). In this scenario, **getpeername**() is used to determine the connecting client's IP address.

getpeername() takes three parameters:

s contains the file descriptor of the socket whose peer should be looked up.

name points to a sockaddr structure that will hold the address information for the connected peer. Normal use requires one to use a structure specific to the protocol family in use, such as sockaddr_in (IPv4) or sockaddr_in6 (IPv6), cast to a (struct sockaddr *).

For greater portability, especially with the newer protocol families, the new struct sockaddr_storage should be used. sockaddr_storage is large enough to hold any of the other sockaddr_* variants. On return, it can be cast to the correct sockaddr type, based on the protocol family contained in its ss_family field.

namelen indicates the amount of space pointed to by *name*, in bytes.

If address information for the local end of the socket is required, the getsockname(2) function should be used instead.

If *name* does not point to enough space to hold the entire socket address, the result will be truncated to *namelen* bytes.

RETURN VALUES

If the call succeeds, a 0 is returned and *namelen* is set to the actual size of the socket address returned in *name*. Otherwise, *errno* is set and a value of −1 is returned.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOTCONN]	The socket is not connected.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *name* parameter points to memory not in a valid part of the process address space.

SEE ALSO

`accept`(2), `bind`(2), `getsockname`(2), `socket`(2)

HISTORY

The **getpeername**() function call appeared in 4.2BSD.

NAME

getpgrp, **getpgid** — get process group

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

pid_t
getpgrp(*void*);

pid_t
getpgid(*pid_t pid*);

DESCRIPTION

The process group of the current process is returned by **getpgrp**(). The process group of the *pid* process is returned by **getpgid**().

Process groups are used for distribution of signals, and by terminals to arbitrate requests for their input: processes that have the same process group as the terminal are foreground and may read, while others will block with a signal if they attempt to read.

This call is thus used by programs such as csh(1) to create process groups in implementing job control. The **tcgetpgrp**() and **tcsetpgrp**() calls are used to get/set the process group of the control terminal.

ERRORS

getpgrp() always succeeds, however **getpgid**() will succeed unless:

[ESRCH]	if there is no process with a process ID equal to *pid*.

SEE ALSO

setpgid(2), termios(4)

STANDARDS

The **getpgrp**() function conforms to IEEE Std 1003.1-1988 (“POSIX.1”).

HISTORY

The **getpgrp**() function call appeared in 4.0BSD. The **getpgid**() function call is derived from its usage in AT&T System V.4 UNIX, and first appeared in NetBSD 1.3.

COMPATIBILITY

This version of **getpgrp**() differs from past Berkeley versions by not taking a *pid_t pid* argument. This incompatibility is required by ISO/IEC 9945-1:1990 (“POSIX.1”).

From the ISO/IEC 9945-1:1990 (“POSIX.1”) Rationale:

4.3BSD provides a **getpgrp**() function that returns the process group ID for a specified process. Although this function is used to support job control, all known job-control shells always specify the calling process with this function. Thus, the simpler System V **getpgrp**() suffices, and the added complexity of the 4.3BSD **getpgrp**() has been omitted from POSIX.1. The old functionality is available from the **getpgid**() function.

NAME

getpid, **getppid** — get parent or calling process identification

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

pid_t
getpid(*void*);

pid_t
getppid(*void*);

DESCRIPTION

getpid() returns the process ID of the calling process. The ID is guaranteed to be unique and is useful for constructing temporary file names.

getppid() returns the process ID of the parent of the calling process.

ERRORS

The **getpid**() and **getppid**() functions are always successful, and no return value is reserved to indicate an error.

SEE ALSO

gethostid(3)

STANDARDS

getpid() and **getppid**() conform to ISO/IEC 9945-1:1990 (“POSIX.1”).

NAME

getpriority, **setpriority** — get/set program scheduling priority

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/resource.h>

int
getpriority(*int which*, *id_t who*);

int
setpriority(*int which*, *id_t who*, *int prio*);

DESCRIPTION

The scheduling priority of the process, process group, or user, as indicated by *which* and *who* is obtained with the **getpriority**() call and set with the **setpriority**() call. *which* is one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER, and *who* is interpreted relative to *which* (a process identifier for PRIO_PROCESS, process group identifier for PRIO_PGRP, and a user ID for PRIO_USER). A zero value of *who* denotes the current process, process group, or user. *prio* is a value in the range -20 to 20. The default priority is 0; lower priorities cause more favorable scheduling. A value of 19 or 20 will schedule a process only when nothing at priority ≤ 0 is runnable.

The **getpriority**() call returns the highest priority (lowest numerical value) enjoyed by any of the specified processes. The **setpriority**() call sets the priorities of all of the specified processes to the specified value. Only the super-user may lower priorities.

RETURN VALUES

Since **getpriority**() can legitimately return the value −1, it is necessary to clear the external variable *errno* prior to the call, then check it afterward to determine if a −1 is an error or a legitimate value. The **setpriority**() call returns 0 if there is no error, or −1 if there is.

ERRORS

getpriority() and **setpriority**() will fail if:

[ESRCH]	No process was located using the *which* and *who* values specified.
[EINVAL]	*which* was not one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER.

In addition to the errors indicated above, **setpriority**() will fail if:

[EPERM]	A process was located, but neither its effective nor real user ID matched the effective user ID of the caller.
[EACCES]	A non super-user attempted to lower a process priority.

SEE ALSO

nice(1), fork(2), renice(8)

HISTORY

The **getpriority**() function call appeared in 4.2BSD.

NAME

getrlimit, **setrlimit** — control maximum system resource consumption

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/resource.h>

int
getrlimit(*int resource*, *struct rlimit *rlp*);

int
setrlimit(*int resource*, *const struct rlimit *rlp*);

DESCRIPTION

Limits on the consumption of system resources by the current process and each process it creates may be obtained with the **getrlimit**() call, and set with the **setrlimit**() call. Resources of an arbitrary process can be obtained/changed using sysctl(3).

The *resource* parameter is one of the following:

RLIMIT_CORE	The largest size (in bytes) core file that may be created.
RLIMIT_CPU	The maximum amount of CPU time (in seconds) to be used by each process.
RLIMIT_DATA	The maximum size (in bytes) of the data segment for a process; this defines how far a program may extend its break with the sbrk(2) system call.
RLIMIT_FSIZE	The largest size (in bytes) file that may be created.
RLIMIT_MEMLOCK	The maximum size (in bytes) which a process may lock into memory using the mlock(2) function.
RLIMIT_NOFILE	The maximum number of open files for this process.
RLIMIT_NPROC	The maximum number of simultaneous processes for this user id.
RLIMIT_RSS	The maximum size (in bytes) to which a process's resident set size may grow. This imposes a limit on the amount of physical memory to be given to a process; if memory is tight, the system will prefer to take memory from processes that are exceeding their declared resident set size.
RLIMIT_SBSIZE	The maximum size (in bytes) of the socket buffers set by the setsockopt(2) SO_RCVBUF and SO_SNDBUF options.
RLIMIT_STACK	The maximum size (in bytes) of the stack segment for a process; this defines how far a program's stack segment may be extended. Stack extension is performed automatically by the system.

A resource limit is specified as a soft limit and a hard limit. When a soft limit is exceeded a process may receive a signal (for example, if the CPU time or file size is exceeded), but it will be allowed to continue execution until it reaches the hard limit (or modifies its resource limit). The *rlimit* structure is used to specify the hard and soft limits on a resource,

```
struct rlimit {
        rlim_t  rlim_cur;      /* current (soft) limit */
        rlim_t  rlim_max;      /* hard limit */
};
```

Only the super-user may raise the maximum limits. Other users may only alter *rlim_cur* within the range from 0 to *rlim_max* or (irreversibly) lower *rlim_max*.

An “infinite” value for a limit is defined as RLIM_INFINITY.

Because this information is stored in the per-process information, this system call must be executed directly by the shell if it is to affect all future processes created by the shell. Thus, shells provide built-in commands to change the limits (**limit** for csh(1), or **ulimit** for sh(1)).

The system refuses to extend the data or stack space when the limits would be exceeded in the normal way: a brk(2) call fails if the data space limit is reached. When the stack limit is reached, the process receives a segmentation fault (SIGSEGV); if this signal is not caught by a handler using the signal stack, this signal will kill the process.

A file I/O operation that would create a file larger that the process’ soft limit will cause the write to fail and a signal SIGXFSZ to be generated; this normally terminates the process, but may be caught. When the soft CPU time limit is exceeded, a signal SIGXCPU is sent to the offending process.

RETURN VALUES

A 0 return value indicates that the call succeeded, changing or returning the resource limit. Otherwise, −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **getrlimit**() and **setrlimit**() will fail if:

[EFAULT]	The address specified for *rlp* is invalid.
[EINVAL]	Specified *resource* was invalid.
[EINVAL]	In the **setrlimit**() call, the specified *rlim_cur* exceeds the specified *rlim_max*.
[EPERM]	The limit specified to **setrlimit**() would have raised the maximum limit value, and the caller is not the super-user.

The **setrlimit**() function may fail if:

[EINVAL]	The limit specified to **setrlimit**() cannot be lowered, because current usage is already higher than the limit.

SEE ALSO

csh(1), sh(1), mlock(2), quotactl(2), setsockopt(2), sigaction(2), sigaltstack(2), sysctl(3)

HISTORY

The **getrlimit**() function call appeared in 4.2BSD.

NAME

getrusage — get information about resource utilization

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/resource.h>
#define     RUSAGE_SELF  0
#define     RUSAGE_CHILDREN   −1
```

int
getrusage(*int who*, *struct rusage *rusage*);

DESCRIPTION

getrusage() returns information describing the resources used by the current process, or all its terminated child processes. The *who* parameter is either RUSAGE_SELF or RUSAGE_CHILDREN. The buffer to which *rusage* points will be filled in with the following structure:

```
struct rusage {
        struct timeval ru_utime; /* user time used */
        struct timeval ru_stime; /* system time used */
        long ru_maxrss;          /* max resident set size */
        long ru_ixrss;           /* integral shared text memory size */
        long ru_idrss;           /* integral unshared data size */
        long ru_isrss;           /* integral unshared stack size */
        long ru_minflt;          /* page reclaims */
        long ru_majflt;          /* page faults */
        long ru_nswap;           /* swaps */
        long ru_inblock;         /* block input operations */
        long ru_oublock;         /* block output operations */
        long ru_msgsnd;          /* messages sent */
        long ru_msgrcv;          /* messages received */
        long ru_nsignals;        /* signals received */
        long ru_nvcsw;           /* voluntary context switches */
        long ru_nivcsw;          /* involuntary context switches */
};
```

The fields are interpreted as follows:

ru_utime	the total amount of time spent executing in user mode.
ru_stime	the total amount of time spent in the system executing on behalf of the process(es).
ru_maxrss	the maximum resident set size used (in kilobytes).
ru_ixrss	an integral value indicating the amount of memory used by the text segment that was also shared among other processes. This value is expressed in units of kilobytes * ticks-of-execution.
ru_idrss	an integral value of the amount of unshared memory residing in the data segment of a process (expressed in units of kilobytes * ticks-of-execution).
ru_isrss	an integral value of the amount of unshared memory residing in the stack segment of a process (expressed in units of kilobytes * ticks-of-execution).

ru_minflt	the number of page faults serviced without any I/O activity; here I/O activity is avoided by reclaiming a page frame from the list of pages awaiting reallocation.
ru_majflt	the number of page faults serviced that required I/O activity.
ru_nswap	the number of times a process was swapped out of main memory.
ru_inblock	the number of times the file system had to perform input.
ru_oublock	the number of times the file system had to perform output.
ru_msgsnd	the number of IPC messages sent.
ru_msgrcv	the number of IPC messages received.
ru_nsignals	the number of signals delivered.
ru_nvcsw	the number of times a context switch resulted due to a process voluntarily giving up the processor before its time slice was completed (usually to await availability of a resource).
ru_nivcsw	the number of times a context switch resulted due to a higher priority process becoming runnable or because the current process exceeded its time slice.

NOTES

The numbers *ru_inblock* and *ru_oublock* account only for real I/O; data supplied by the caching mechanism is charged only to the first process to read or write the data.

ERRORS

getrusage() returns −1 on error. The possible errors are:

[EINVAL]	The *who* parameter is not a valid value.
[EFAULT]	The address specified by the *rusage* parameter is not in a valid part of the process address space.

SEE ALSO

gettimeofday(2), wait(2)

HISTORY

The **getrusage**() function call appeared in 4.2BSD.

BUGS

There is no way to obtain information about a child process that has not yet terminated.

NAME

getsid — get session ID

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

pid_t
getsid(*pid_t pid*);

DESCRIPTION

The **getsid** function returns the session ID of the process specified by *pid*. If *pid* is 0, the session ID of the calling process is returned. The session ID is defined as the process group ID of the process that is the session leader.

ERRORS

If an error occurs, **getsid** returns −1 and the global variable *errno* is set to indicate the error, as follows:

[ESRCH]	No process can be found corresponding to that specified by *pid*.

SEE ALSO

getpgid(2), setsid(2), termios(4)

STANDARDS

The **getsid**() function conforms to X/Open Portability Guide Issue 4, Version 2 (“XPG4.2”).

NAME

getsockname — get socket name

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/socket.h>

int
getsockname(*int s*, *struct sockaddr * restrict name*,
    *socklen_t * restrict namelen*);

DESCRIPTION

getsockname() returns the locally bound address information for a specified socket.

Common uses of this function are as follows:

- When `bind`(2) is called with a port number of 0 (indicating the kernel should pick an ephemeral port) **getsockname**() is used to retrieve the kernel-assigned port number.
- When a process calls `bind`(2) on a wildcard IP address, **getsockname**() is used to retrieve the local IP address for the connection.
- When a function wishes to know the address family of a socket, **getsockname**() can be used.

getsockname() takes three parameters:

s, Contains the file descriptor for the socket to be looked up.

name points to a `sockaddr` structure which will hold the resulting address information. Normal use requires one to use a structure specific to the protocol family in use, such as `sockaddr_in` (IPv4) or `sockaddr_in6` (IPv6), cast to a (struct sockaddr *).

For greater portability (such as newer protocol families) the new structure sockaddr_storage exists. `sockaddr_storage` is large enough to hold any of the other sockaddr_* variants. On return, it should be cast to the correct sockaddr type, according to the current protocol family.

namelen indicates the amount of space pointed to by *name*, in bytes. Upon return, *namelen* is set to the actual size of the returned address information.

If the address of the destination socket for a given socket connection is needed, the `getpeername`(2) function should be used instead.

If *name* does not point to enough space to hold the entire socket address, the result will be truncated to *namelen* bytes.

RETURN VALUES

On success, **getsockname**() returns a 0, and *namelen* is set to the actual size of the socket address returned in *name*. Otherwise, *errno* is set, and a value of −1 is returned.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.

`[EINVAL]`	The socket has been shut down.
`[ENOBUFS]`	Insufficient resources were available in the system to perform the operation.
`[EFAULT]`	The *name* parameter points to memory not in a valid part of the process address space.

SEE ALSO

`bind`(2), `socket`(2)

HISTORY

The **getsockname**() function call appeared in 4.2BSD.

BUGS

Names bound to sockets in the UNIX domain are inaccessible; **getsockname**() returns a zero length name.

NAME

getsockopt, **setsockopt** — get and set options on sockets

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/socket.h>

int
getsockopt(*int s*, *int level*, *int optname*, *void * restrict optval*,
    *socklen_t * restrict optlen*);

int
setsockopt(*int s*, *int level*, *int optname*, *const void *optval*,
    socklen_t optlen);

DESCRIPTION

getsockopt() and **setsockopt**() manipulate the *options* associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost “socket” level.

When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the socket level, *level* is specified as SOL_SOCKET. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, *level* should be set to the protocol number of TCP; see getprotoent(3).

The parameters *optval* and *optlen* are used to access option values for **setsockopt**(). For **getsockopt**() they identify a buffer in which the value for the requested option(s) are to be returned. For **getsockopt**(), *optlen* is a value-result parameter, initially containing the size of the buffer pointed to by *optval*, and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, *optval* may be NULL.

optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file ⟨sys/socket.h⟩ contains definitions for socket level options, described below. Options at other protocol levels vary in format and name; consult the appropriate entries in section 4 of the manual, including: clnp(4), faith(4), icmp6(4), ip(4), ip6(4), ipsec(4), multicast(4), pim(4), route(4), tcp(4), tp(4), and unix(4).

Most socket-level options use an *int* parameter for *optval*. For **setsockopt**(), the parameter should be non-zero to enable a boolean option, or zero if the option is to be disabled. SO_LINGER uses a *struct linger* parameter, defined in ⟨sys/socket.h⟩, which specifies the desired state of the option and the linger interval (see below). SO_SNDTIMEO and SO_RCVTIMEO use a *struct timeval* parameter, defined in ⟨sys/time.h⟩.

The following options are recognized at the socket level. Except as noted, each may be examined with **getsockopt**() and set with **setsockopt**().

SO_DEBUG	enables recording of debugging information
SO_REUSEADDR	enables local address reuse
SO_REUSEPORT	enables duplicate address and port bindings
SO_KEEPALIVE	enables keep connections alive
SO_DONTROUTE	enables routing bypass for outgoing messages

`SO_LINGER`	linger on close if data present
`SO_BROADCAST`	enables permission to transmit broadcast messages
`SO_OOBINLINE`	enables reception of out-of-band data in band
`SO_SNDBUF`	set buffer size for output
`SO_RCVBUF`	set buffer size for input
`SO_SNDLOWAT`	set minimum count for output
`SO_RCVLOWAT`	set minimum count for input
`SO_SNDTIMEO`	set timeout value for output
`SO_RCVTIMEO`	set timeout value for input
`SO_TIMESTAMP`	enables reception of a timestamp with datagrams
`SO_TYPE`	get the type of the socket (get only)
`SO_ERROR`	get and clear error on the socket (get only)

`SO_DEBUG` enables debugging in the underlying protocol modules. `SO_REUSEADDR` indicates that the rules used in validating addresses supplied in a `bind`(2) call should allow reuse of local addresses. `SO_REUSEPORT` allows completely duplicate bindings by multiple processes if they all set `SO_REUSEPORT` before binding the port. This option permits multiple instances of a program to each receive UDP/IP multicast or broadcast datagrams destined for the bound port. `SO_KEEPALIVE` enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified via a `SIGPIPE` signal when attempting to send data. `SO_DONTROUTE` indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

`SO_LINGER` controls the action taken when unsent messages are queued on socket and a `close`(2) is performed. If the socket promises reliable delivery of data and `SO_LINGER` is set, the system will block the process on the `close`(2) attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, measured in seconds, termed the linger interval, is specified in the **setsockopt**() call when `SO_LINGER` is requested). If `SO_LINGER` is disabled and a `close`(2) is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

The option `SO_BROADCAST` requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the system. With protocols that support out-of-band data, the `SO_OOBINLINE` option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with `recv`(2) or `read`(2) calls without the `MSG_OOB` flag. Some protocols always behave as if this option is set. `SO_SNDBUF` and `SO_RCVBUF` are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute limit on these values.

`SO_SNDLOWAT` is an option to set the minimum count for output operations. Most output operations process all of the data supplied by the call, delivering data to the protocol for transmission and blocking as necessary for flow control. Nonblocking output operations will process as much data as permitted subject to flow control without blocking, but will process no data if flow control does not allow the smaller of the low water mark value or the entire request to be processed. A `select`(2) or `poll`(2) operation testing the ability to write to a socket will return true only if the low water mark amount could be processed. The default value for `SO_SNDLOWAT` is set to a convenient size for network efficiency, often 1024. `SO_RCVLOWAT` is an option to set the minimum count for input operations. In general, receive calls will block until any (non-zero) amount of data is received, then return with the smaller of the amount available or the amount requested. The default value for `SO_RCVLOWAT` is 1. If `SO_RCVLOWAT` is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. Receive calls may still return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different than that returned.

`SO_SNDTIMEO` is an option to set a timeout value for output operations. It accepts a *`struct timeval`* parameter with the number of seconds and microseconds used to limit waits for output operations to complete. If a send operation has blocked for this much time, it returns with a partial count or with the error `EAGAIN` if no data were sent. In the current implementation, this timer is restarted each time additional data are delivered to the protocol, implying that the limit applies to output portions ranging in size from the low water mark to the high water mark for output. `SO_RCVTIMEO` is an option to set a timeout value for input operations. It accepts a *`struct timeval`* parameter with the number of seconds and microseconds used to limit waits for input operations to complete. In the current implementation, this timer is restarted each time additional data are received by the protocol, and thus the limit is in effect an inactivity timer. If a receive operation has been blocked for this much time without receiving additional data, it returns with a short count or with the error `EAGAIN` if no data were received.

If the `SO_TIMESTAMP` option is enabled on a `SOCK_DGRAM` socket, the `recvmsg`(2) call will return a timestamp corresponding to when the datagram was received. The msg_control field in the msghdr structure points to a buffer that contains a cmsghdr structure followed by a struct timeval. The cmsghdr fields have the following values:

```
cmsg_len = sizeof(struct timeval)
cmsg_level = SOL_SOCKET
cmsg_type = SCM_TIMESTAMP
```

Finally, `SO_TYPE` and `SO_ERROR` are options used only with **`getsockopt`**(). `SO_TYPE` returns the type of the socket, such as `SOCK_STREAM`; it is useful for servers that inherit sockets on startup. `SO_ERROR` returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

RETURN VALUES

A 0 is returned if the call succeeds, −1 if it fails.

ERRORS

The call succeeds unless:

`[EBADF]`	The argument *`s`* is not a valid descriptor.
`[ENOTSOCK]`	The argument *`s`* is a file, not a socket.
`[ENOPROTOOPT]`	The option is unknown at the level indicated.
`[EFAULT]`	The address pointed to by *`optval`* is not in a valid part of the process address space. For **`getsockopt`**(), this error may also be returned if *`optlen`* is not in a valid part of the process address space.

SEE ALSO

`ioctl`(2), `poll`(2), `select`(2), `socket`(2), `getprotoent`(3), `clnp`(4), `faith`(4), `icmp6`(4), `ip`(4), `ip6`(4), `ipsec`(4), `multicast`(4), `pim`(4), `route`(4), `tcp`(4), `tp`(4), `unix`(4), `protocols`(5)

HISTORY

The **`getsockopt`**() system call appeared in 4.2BSD.

BUGS

Several of the socket options should be handled at lower levels of the system.

NAME

gettimeofday, **settimeofday** — get/set date and time

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/time.h>

int
gettimeofday(*struct timeval ∗ restrict tp*, *void ∗ restrict tzp*);

int
settimeofday(*const struct timeval ∗ restrict tp*,
    const void ∗ restrict tzp);

DESCRIPTION

Note: time zone information is no longer provided by this interface. See `localtime`(3) for information on how to retrieve it.

The system's notion of the current UTC time is obtained with the **gettimeofday**() call, and set with the **settimeofday**() call. The time is expressed in seconds and microseconds since midnight (0 hour), January 1, 1970. The resolution of the system clock is hardware dependent, and the time may be updated continuously or in "ticks." If *tp* is NULL, the time will not be returned or set. Despite being declared *void ∗*, the objects pointed to by *tzp* shall be of type *struct timezone*.

The structures pointed to by *tp* and *tzp* are defined in ⟨`sys/time.h`⟩ as:

```
struct timeval {
        long    tv_sec;         /* seconds since Jan. 1, 1970 */
        long    tv_usec;        /* and microseconds */
};

struct timezone {
        int     tz_minuteswest; /* of Greenwich */
        int     tz_dsttime;     /* type of dst correction to apply */
};
```

The *timezone* structure is provided only for source compatibility. It is ignored by **settimeofday**(), and **gettimeofday**() will always return zeroes.

If the calling user is not the super-user, then the **settimeofday**() function in the standard C library will try to use the `clockctl`(4) device if present, thus making possible for non privileged users to set the system time. If `clockctl`(4) is not present or not accessible, then **settimeofday**() reverts to the **settimeofday**() system call, which is restricted to the super user.

RETURN VALUES

A 0 return value indicates that the call succeeded. A −1 return value indicates an error occurred, and in this case an error code is stored into the global variable *errno*.

ERRORS

The following error codes may be set in *errno*:

`[EFAULT]`	An argument address referenced invalid memory.

`[EPERM]`	A user other than the super user attempted to set the time, or the specified time was less than the current time, which was not permitted at the current security level.

SEE ALSO

`date`(1), `adjtime`(2), `ctime`(3), `localtime`(3), `clockctl`(4), `timed`(8)

HISTORY

The **`gettimeofday`**() function call appeared in 4.2BSD. The *`tzp`* argument was deprecated in 4.4BSD (and many other systems).

NAME

getuid, **geteuid** — get user identification

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

uid_t
getuid(*void*);

uid_t
geteuid(*void*);

DESCRIPTION

The **getuid**() function returns the real user ID of the calling process. The **geteuid**() function returns the effective user ID of the calling process.

The real user ID is that of the user who has invoked the program. As the effective user ID gives the process additional permissions during execution of "*set-user-ID*" mode processes, **getuid**() is used to determine the real-user-id of the calling process.

ERRORS

The **getuid**() and **geteuid**() functions are always successful, and no return value is reserved to indicate an error.

SEE ALSO

getgid(2), setreuid(2)

STANDARDS

The **geteuid**() and **getuid**() functions conform to ISO/IEC 9945-1:1990 ("POSIX.1").

NAME

getvfsstat — get list of all mounted file systems

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/statvfs.h>

int
getvfsstat(struct statvfs *buf, size_t bufsize, int flags);
```

DESCRIPTION

getvfsstat() returns information about all mounted file systems. *buf* is a pointer to an array of **statvfs** structures defined in `statvfs`(5).

The buffer is filled with an array of *statvfs* structures, one for each mounted file system up to the size specified by *bufsize*.

If *buf* is given as `NULL`, **getvfsstat**() returns just the number of mounted file systems.

Normally *flags* should be specified as `ST_WAIT`. If *flags* is set to `ST_NOWAIT`, **getvfsstat**() will return the information it has available without requesting an update from each file system. Thus, some of the information will be out of date, but **getvfsstat**() will not block waiting for information from a file system that is unable to respond.

RETURN VALUES

Upon successful completion, the number of *statvfs* structures is returned. Otherwise, −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

getvfsstat() fails if one or more of the following are true:

[EFAULT]	*buf* points to an invalid address.
[EIO]	An I/O error occurred while reading from or writing to the file system.

SEE ALSO

`statvfs`(2), `getmntinfo`(3), `fstab`(5), `mount`(8)

HISTORY

The **getvfsstat**() function first appeared in NetBSD 3.0 to replace **getfsstat**() which appeared in 4.4BSD.

NAME

i386_get_ldt, **i386_set_ldt** — manage i386 per-process Local Descriptor Table entries

LIBRARY

i386 Architecture Library (libi386, −li386)

SYNOPSIS

#include <sys/types.h>
#include <machine/segments.h>
#include <machine/sysarch.h>

int
i386_get_ldt(*int start_sel*, *union descriptor *descs*, *int num_sels*);

int
i386_set_ldt(*int start_sel*, *union descriptor *descs*, *int num_sels*);

DESCRIPTION

i386_get_ldt() will return the list of i386 descriptors that the process has in its LDT. **i386_set_ldt**() will set a list of i386 descriptors for the current process in its LDT. Both routines accept a starting selector number *start_sel* , an array of memory that will contain the descriptors to be set or returned *descs* , and the number of entries to set or return *num_sels*.

The argument *descs* can be either segment_descriptor or gate_descriptor and are defined in **<i386/segments.h> .**
These structures are defined by the architecture as disjoint bit-fields, so care must be taken in constructing them.

RETURN VALUES

Upon successful completion, **i386_get_ldt**() returns the number of descriptors currently in the LDT. **i386_set_ldt**() returns the first selector set. Otherwise, a value of −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

i386_get_ldt() and **i386_set_ldt**() will fail if:

[EINVAL]	An inappropriate parameter was used for *start_sel* or *num_sels*.
[EACCES]	The caller attempted to use a descriptor that would circumvent protection or cause a failure.

REFERENCES

i386 Microprocessor Programmer’s Reference Manual, Intel

WARNING

You can really hose your process using this.

NAME

i386_get_mtrr, **i386_set_mtrr** — access Memory Type Range Registers

LIBRARY

i386 Architecture Library (libi386, −li386)

SYNOPSIS

#include <sys/types.h>
#include <machine/sysarch.h>
#include <machine/mtrr.h>

int
i386_get_mtrr(*struct mtrr *mtrrp*, *int *n*);

int
i386_set_mtrr(*struct mtrr *mtrrp*, *int *n*);

DESCRIPTION

These functions provide an interface to the MTRR registers found on 686-class processors for controlling processor access to memory ranges. This is most useful for accessing devices such as video accelerators on pci(4) and agp(4) buses. For example, enabling write-combining allows bus-write transfers to be combined into a larger transfer before bursting over the bus. This can increase performance of write operations 2.5 times or more.

mtrrp is a pointer to one or more mtrr structures, as described below. The *n* argument is a pointer to an integer containing the number of structures pointed to by *mtrrp*. For **i386_set_mtrr**() the integer pointed to by *n* will be updated to reflect the actual number of MTRRs successfully set. For **i386_get_mtrr**() no more than *n* structures will be copied out, and the integer value pointed to by *n* will be updated to reflect the actual number of valid structures retrieved. A NULL argument to *mtrrp* will result in just the number of MTRRs available being returned in the integer pointed to by *n*.

The argument *mtrrp* has the following structure:

```
struct mtrr {
        uint64_t base;
        uint64_t len;
        uint8_t type;
        int flags;
        pid_t owner;
};
```

The location of the mapping is described by its physical base address *base* and length *len*. Valid values for *type* are:

MTRR_TYPE_UC	uncached memory
MTRR_TYPE_WC	use write-combining
MTRR_TYPE_WT	use write-through caching
MTRR_TYPE_WP	write-protected memory
MTRR_TYPE_WB	use write-back caching

Valid values for *flags* are:

MTRR_PRIVATE
own range, reset the MTRR when the current process exits

MTRR_FIXED	use fixed range MTRR
MTRR_VALID	entry is valid

The *owner* member is the PID of the user process which claims the mapping. It is only valid if MTRR_PRIVATE is set in *flags*. To clear/reset MTRRs, use a *flags* field without MTRR_VALID set.

RETURN VALUES

Upon successful completion zero is returned, otherwise −1 is returned on failure, and the global variable *errno* is set to indicate the error. The integer value pointed to by *n* will contain the number of successfully processed mtrr structures in both cases.

ERRORS

`[ENOSYS]`	The currently running kernel or CPU has no MTRR support.
`[EINVAL]`	The currently running kernel has no MTRR support, or one of the mtrr structures pointed to by *mtrrp* is invalid.
`[EBUSY]`	No unused MTRRs are available.

HISTORY

The **i386_get_mtrr**() and **i386_set_mtrr**() functions appeared in NetBSD 1.6.

NAME

i386_iopl — change the i386 I/O privilege level

LIBRARY

i386 Architecture Library (libi386, −li386)

SYNOPSIS

#include <sys/types.h>
#include <machine/sysarch.h>

int
i386_iopl(*int iopl*);

DESCRIPTION

i386_iopl() sets the i386 I/O privilege level to the value specified by *iopl*. This call is restricted to the super-user.

RETURN VALUES

Upon successful completion, **i386_iopl**() returns 0. Otherwise, a value of −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

i386_iopl() will fail if:

[EPERM] The caller was not the super-user, or the operation was not permitted at the current security level.

REFERENCES

i386 Microprocessor Programmer's Reference Manual, Intel

WARNING

You can really hose your machine if you enable user-level I/O and write to hardware ports without care.

NAME

i386_pmc_info, **i386_pmc_startstop**, **i386_pmc_read** — interface to CPU performance counters

LIBRARY

i386 Architecture Library (libi386, −li386)

SYNOPSIS

#include <sys/types.h>
#include <machine/sysarch.h>
#include <machine/specialreg.h>

int
i386_pmc_info(*struct i386_pmc_info_args ∗ia*);

int
i386_pmc_startstop(*struct i386_pmc_startstop_args ∗ssa*);

int
i386_pmc_read(*struct i386_pmc_read_args ∗ra*);

DESCRIPTION

These functions provide an interface to the CPU performance counters on the 586-class and 686-class processors.

i386_pmc_info() will return information about the available CPU counters. The information is returned in *ia* having the following structure:

```
struct i386_pmc_info_args {
        int     type;
        int     flags;
};
```

The *type* member describes the class of performance counters available. Valid values are:

PMC_TYPE_NONE	No PMC support
PMC_TYPE_I586	586-class CPUs
PMC_TYPE_I686	686-class Intel CPUs
PMC_TYPE_K7	686-class AMD CPUs

The *flags* member describes additional capabilities of the processor. Valid values are:

PMC_INFO_HASTSC	CPU has time-stamp counter

i386_pmc_startstop() is used to start and stop the measurement of the CPU performance counters. The argument *ssa* has the following structure:

```
struct i386_pmc_startstop_args {
        int counter;
        uint64_t val;
        uint8_t event;
        uint8_t unit;
        uint8_t compare;
        uint8_t flags;
};
```

The counter specified by the member *counter* is started if the member *flags* has PMC_SETUP_KERNEL or PMC_SETUP_USER set, otherwise the counter is stopped. The initial value of the counter is set to *val*. Additional values for the *flags* member are PMC_SETUP_EDGE and PMC_SETUP_INV. The *event* member specifies some event written to the control register. The *unit* member specifies the measurement units. The *compare* member is a mask for the counter.

i386_pmc_read() will return information about a specific CPU counter measured during the last measurement period determined by the calling of **i386_pmc_startstop**(). The information is returned in *ra* having the following structure:

```
struct i386_pmc_read_args {
        int counter;
        uint64_t val;
        uint64_t time;
};
```

The counter to read should be specified by the *counter* member. Counters are numbered from 0 to PMC_NCOUNTERS. The value of the counter is returned in the *val* member. The time since epoch, measured in CPU clock cycles, is returned in the *time* member.

RETURN VALUES

Upon successful completion zero is returned, otherwise −1 is returned on failure.

NAME

i386_vm86 — set virtual 8086 processor registers and mode

LIBRARY

i386 Architecture Library (libi386, −li386)

SYNOPSIS

#include <sys/types.h>
#include <signal.h>
#include <machine/mcontext.h>
#include <machine/segments.h>
#include <machine/sysarch.h>
#include <machine/vm86.h>

int
i386_vm86(*struct vm86_struct *vmcp*);

DESCRIPTION

i386_vm86() will set the process into virtual 8086 mode using the registers and selectors specified by the context pointed to by *vmcp*. The processor registers are set from *vmcp->substr.regs*, and the emulated processor type from *vmcp->substr.ss_cpu_type*.

The kernel keeps a pointer to the context, and uses the tables stored at *vmcp->int_byuser* and *vmcp->int21_byuser* for fast virtual interrupt handling. If the *n th* bit is clear in the first of these arrays, then the kernel may directly emulate the real-mode x86 INT *n* instruction handling. If the *n* th bit is set, then the process is delivered a signal when an INT instruction is executed.

Since MS-DOS puts many DOS functions onto interrupt 21, it is handled specially: the *k* th bit in the *vmcp->int21_byuser* array is checked when INT *21* is requested and the *ah* register is *k*.

RETURN VALUES

This routine does not normally return: 32-bit mode will be restored by the delivery of a signal to the process. In case of an error in setting the VM86 mode, a value of −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

i386_vm86() will fail if:

[EFAULT] The state at *vmcp* was not readable to the user process.

REFERENCES

i386 Microprocessor Programmer's Reference Manual, Intel

NAME

ioctl — control device

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/ioctl.h>

int
ioctl(*int d*, *unsigned long request*, *void *argp*);

DESCRIPTION

The **ioctl**() function manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g. terminals) may be controlled with **ioctl**() requests. The argument *d* must be an open file descriptor.

An ioctl *request* has encoded in it whether the argument is an "in" parameter or "out" parameter, and the size of the argument *argp* in bytes. Macros and defines used in specifying an ioctl *request* are located in the file ⟨sys/ioctl.h⟩.

GENERIC IOCTLS

Some ioctls are applicable to any file descriptor. These include:

FIOCLEX
: Set close-on-exec flag. The file will be closed when exec(3) is invoked.

FIONCLEX
: Clear close-on-exec flag. The file will remain open across exec(3).

Some generic ioctls are not implemented for all types of file descriptors. These include:

FIONREAD int
: Get the number of bytes that are immediately available for reading.

FIONWRITE int
: Get the number of bytes in the descriptor's send queue. These bytes are data which has been written to the descriptor but which are being held by the kernel for further processing. The nature of the required processing depends on the underlying device. For tty devices, these bytes are typically queued for delivery to the tty hardware. For TCP sockets, these bytes have not yet been acknolwedged by the other side of the connection. For files, this operation always returns zero as files do not have send queues.

FIONSPACE int
: Get the free space in the descriptor's send queue. This value is the size of the send queue minus the number of bytes being held in the queue. Note: while this value represents the number of bytes that may be added to the queue, other resource limitations may cause a write not larger than the send queue's space to be blocked. One such limitation would be a lack of network buffers for a write to a network connection.

FIONBIO int
: Set non-blocking I/O mode if the argument is non-zero. In non-blocking mode, read(2) or write(2) calls return −1 and set *errno* to EAGAIN immediately when no data is available.

FIOASYNC int
: Set asynchronous I/O mode if the argument is non-zero. In asynchronous mode, the process or process group specified by FIOSETOWN will start receiving SIGIO signals when data is available.

The `SIGIO` signal will be delivered when data is available on the file descriptor.

`FIOSETOWN, FIOGETOWN int`
Set/get the process or the process group (if negative) that should receive `SIGIO` signals when data is available.

RETURN VALUES

If an error has occurred, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

ioctl() will fail if:

[EBADF]	*d* is not a valid descriptor.
[ENOTTY]	*d* is not associated with a character special device.
[ENOTTY]	The specified request does not apply to the kind of object that the descriptor *d* references.
[EINVAL]	*request* or *argp* is not valid.
[EFAULT]	*argp* points outside the process's allocated address space.

SEE ALSO

`mt`(1), `execve`(2), `fcntl`(2), `intro`(4), `tty`(4)

HISTORY

An **ioctl**() function call appeared in Version 7 AT&T UNIX.

NAME

issetugid — is current process tainted by uid or gid changes

SYNOPSIS

#include <unistd.h>

int
issetugid(*void*);

DESCRIPTION

The **issetugid**() function returns 1 if the process environment or memory address space is considered "tainted", and returns 0 otherwise.

A process is tainted if it was created as a result of an execve(2) system call which had either of the setuid or setgid bits set (and extra privileges were given as a result) or if it has changed any of its real, effective or saved user or group ID's since it began execution.

This system call exists so that library routines (e.g., libc, libtermcap) can reliably determine if it is safe to use information that was obtained from the user, in particular the results from getenv(3) should be viewed with suspicion if it is used to control operation.

A "tainted" status is inherited by child processes as a result of the fork(2) system call (or other library code that calls fork, such as popen(3)).

It is assumed that a program that clears all privileges as it prepares to execute another will also reset the environment, hence the "tainted" status will not be passed on. This is important for programs such as su(1) which begin setuid but need to be able to create an untainted process.

ERRORS

The **issetugid**() function is always successful, and no return value is reserved to indicate an error.

SEE ALSO

execve(2), fork(2), setegid(2), seteuid(2), setgid(2), setregid(2), setreuid(2), setuid(2)

HISTORY

A **issetugid**() function call first appeared in OpenBSD 2.0 and was also implemented in FreeBSD 3.0. FreeBSD implementation was imported in NetBSD 1.5.

NAME

kill — send signal to a process

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <signal.h>

int
kill(*pid_t pid*, *int sig*);

DESCRIPTION

The **kill**() function sends the signal given by *sig* to *pid*, a process or a group of processes. *sig* may be one of the signals specified in sigaction(2) or it may be 0, in which case error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

For a process to have permission to send a signal to a process designated by *pid*, the real or effective user ID of the receiving process must match that of the sending process or the user must have appropriate privileges (such as given by a set-user-ID program or the user is the super-user). A single exception is the signal SIGCONT, which may always be sent to any descendant of the current process.

If *pid* is greater than zero:
sig is sent to the process whose ID is equal to *pid*.

If *pid* is zero:
sig is sent to all processes whose group ID is equal to the process group ID of the sender, and for which the process has permission; this is a variant of killpg(3).

If *pid* is −1:
If the user has super-user privileges, the signal is sent to all processes excluding system processes and the process sending the signal. If the user is not the super user, the signal is sent to all processes with the same uid as the user excluding the process sending the signal. No error is returned if any process could be signaled.

For compatibility with System V, if the process number is negative but not −1, the signal is sent to all processes whose process group ID is equal to the absolute value of the process number. This is a variant of killpg(3).

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

kill() will fail and no signal will be sent if:

[EINVAL]	*sig* is not a valid signal number.
[ESRCH]	No process can be found corresponding to that specified by *pid*.
[ESRCH]	The process id was given as 0 but the sending process does not have a process group.
[EPERM]	The sending process is not the super-user and its effective user id does not match the effective user-id of the receiving process. When signaling a process group, this error is returned if any members of the group could not be signaled.

SEE ALSO

`getpgrp`(2), `getpid`(2), `sigaction`(2), `killpg`(3), `signal`(7)

STANDARDS

The **`kill`**() function is expected to conform to ISO/IEC 9945-1:1990 (“POSIX.1”).

NAME

kqueue, **kevent** — kernel event notification mechanism

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/event.h>
#include <sys/time.h>

int
kqueue(*void*);

int
kevent(*int kq*, *const struct kevent *changelist*, *size_t nchanges*,
    *struct kevent *eventlist*, *size_t nevents*,
    *const struct timespec *timeout*);

EV_SET(*&kev*, *ident*, *filter*, *flags*, *fflags*, *data*, *udata*);

DESCRIPTION

kqueue() provides a generic method of notifying the user when an event happens or a condition holds, based on the results of small pieces of kernel code termed filters. A kevent is identified by the (ident, filter) pair; there may only be one unique kevent per kqueue.

The filter is executed upon the initial registration of a kevent in order to detect whether a preexisting condition is present, and is also executed whenever an event is passed to the filter for evaluation. If the filter determines that the condition should be reported, then the kevent is placed on the kqueue for the user to retrieve.

The filter is also run when the user attempts to retrieve the kevent from the kqueue. If the filter indicates that the condition that triggered the event no longer holds, the kevent is removed from the kqueue and is not returned.

Multiple events which trigger the filter do not result in multiple kevents being placed on the kqueue; instead, the filter will aggregate the events into a single struct kevent. Calling **close**() on a file descriptor will remove any kevents that reference the descriptor.

kqueue() creates a new kernel event queue and returns a descriptor. The queue is not inherited by a child created with fork(2).

kevent() is used to register events with the queue, and return any pending events to the user. *changelist* is a pointer to an array of *kevent* structures, as defined in ⟨sys/event.h⟩. All changes contained in the *changelist* are applied before any pending events are read from the queue. *nchanges* gives the size of *changelist*. *eventlist* is a pointer to an array of kevent structures. *nevents* determines the size of *eventlist*. If *timeout* is a non-NULL pointer, it specifies a maximum interval to wait for an event, which will be interpreted as a struct timespec. If *timeout* is a NULL pointer, **kevent**() waits indefinitely. To effect a poll, the *timeout* argument should be non-NULL, pointing to a zero-valued *timespec* structure. The same array may be used for the *changelist* and *eventlist*.

EV_SET() is a macro which is provided for ease of initializing a kevent structure.

The *kevent* structure is defined as:

```
struct kevent {
        uintptr_t ident;       /* identifier for this event */
        uint32_t  filter;      /* filter for event */
        uint32_t  flags;       /* action flags for kqueue */
        uint32_t  fflags;      /∗ filter flag value ∗/
        int64_t   data;               /∗ filter data value ∗/
        intptr_t  udata;      /∗ opaque user data identifier ∗/
};
```

The fields of `struct kevent` are:

ident	Value used to identify this event. The exact interpretation is determined by the attached filter, but often is a file descriptor.
filter	Identifies the kernel filter used to process this event. There are pre-defined system filters (which are described below), and other filters may be added by kernel subsystems as necessary.
flags	Actions to perform on the event.
fflags	Filter-specific flags.
data	Filter-specific data value.
udata	Opaque user-defined value passed through the kernel unchanged.

The *flags* field can contain the following values:

EV_ADD	Adds the event to the kqueue. Re-adding an existing event will modify the parameters of the original event, and not result in a duplicate entry. Adding an event automatically enables it, unless overridden by the EV_DISABLE flag.
EV_ENABLE	Permit **kevent**() to return the event if it is triggered.
EV_DISABLE	Disable the event so **kevent**() will not return it. The filter itself is not disabled.
EV_DELETE	Removes the event from the kqueue. Events which are attached to file descriptors are automatically deleted on the last close of the descriptor.
EV_ONESHOT	Causes the event to return only the first occurrence of the filter being triggered. After the user retrieves the event from the kqueue, it is deleted.
EV_CLEAR	After the event is retrieved by the user, its state is reset. This is useful for filters which report state transitions instead of the current state. Note that some filters may automatically set this flag internally.
EV_EOF	Filters may set this flag to indicate filter-specific EOF condition.
EV_ERROR	See **RETURN VALUES** below.

Filters

Filters are identified by a number. There are two types of filters; pre-defined filters which are described below, and third-party filters that may be added with `kfilter_register`(9) by kernel sub-systems, third-party device drivers, or loadable kernel modules.

As a third-party filter is referenced by a well-known name instead of a statically assigned number, two `ioctl`(2)s are supported on the file descriptor returned by **kqueue**() to map a filter name to a filter number, and vice-versa (passing arguments in a structure described below):

KFILTER_BYFILTER	Map *filter* to *name*, which is of size *len*.
KFILTER_BYNAME	Map *name* to *filter*. *len* is ignored.

The following structure is used to pass arguments in and out of the `ioctl`(2):

```
struct kfilter_mapping {
        char    *name;         /* name to lookup or return */
        size_t   len;          /* length of name */
        uint32_t filter;       /* filter to lookup or return */
};
```

Arguments may be passed to and from the filter via the *fflags* and *data* fields in the kevent structure.

The predefined system filters are:

EVFILT_READ Takes a descriptor as the identifier, and returns whenever there is data available to read. The behavior of the filter is slightly different depending on the descriptor type.

Sockets

Sockets which have previously been passed to **listen**() return when there is an incoming connection pending. *data* contains the size of the listen backlog (i.e., the number of connections ready to be accepted with `accept`(2).)

Other socket descriptors return when there is data to be read, subject to the `SO_RCVLOWAT` value of the socket buffer. This may be overridden with a per-filter low water mark at the time the filter is added by setting the NOTE_LOWAT flag in *fflags*, and specifying the new low water mark in *data*. On return, *data* contains the number of bytes in the socket buffer.

If the read direction of the socket has shutdown, then the filter also sets EV_EOF in *flags*, and returns the socket error (if any) in *fflags*. It is possible for EOF to be returned (indicating the connection is gone) while there is still data pending in the socket buffer.

Vnodes

Returns when the file pointer is not at the end of file. *data* contains the offset from current position to end of file, and may be negative.

Fifos, Pipes

Returns when the there is data to read; *data* contains the number of bytes available.

When the last writer disconnects, the filter will set EV_EOF in *flags*. This may be cleared by passing in EV_CLEAR, at which point the filter will resume waiting for data to become available before returning.

EVFILT_WRITE Takes a descriptor as the identifier, and returns whenever it is possible to write to the descriptor. For sockets, pipes, fifos, and ttys, *data* will contain the amount of space remaining in the write buffer. The filter will set EV_EOF when the reader disconnects, and for the fifo case, this may be cleared by use of EV_CLEAR. Note that this filter is not supported for vnodes.

For sockets, the low water mark and socket error handling is identical to the EVFILT_READ case.

EVFILT_AIO This is not implemented in NetBSD.

EVFILT_VNODE Takes a file descriptor as the identifier and the events to watch for in *fflags*, and returns when one or more of the requested events occurs on the descriptor. The events to monitor are:

NOTE_DELETE **unlink**() was called on the file referenced by the descriptor.

NOTE_WRITE	A write occurred on the file referenced by the descriptor.
NOTE_EXTEND	The file referenced by the descriptor was extended.
NOTE_ATTRIB	The file referenced by the descriptor had its attributes changed.
NOTE_LINK	The link count on the file changed.
NOTE_RENAME	The file referenced by the descriptor was renamed.
NOTE_REVOKE	Access to the file was revoked via `revoke`(2) or the underlying fileystem was unmounted.

On return, *fflags* contains the events which triggered the filter.

EVFILT_PROC Takes the process ID to monitor as the identifier and the events to watch for in *fflags*, and returns when the process performs one or more of the requested events. If a process can normally see another process, it can attach an event to it. The events to monitor are:

NOTE_EXIT	The process has exited.
NOTE_FORK	The process has called **fork**().
NOTE_EXEC	The process has executed a new process via `execve`(2) or similar call.
NOTE_TRACK	Follow a process across **fork**() calls. The parent process will return with NOTE_TRACK set in the *fflags* field, while the child process will return with NOTE_CHILD set in *fflags* and the parent PID in *data*.
NOTE_TRACKERR	This flag is returned if the system was unable to attach an event to the child process, usually due to resource limitations.

On return, *fflags* contains the events which triggered the filter.

EVFILT_SIGNAL Takes the signal number to monitor as the identifier and returns when the given signal is delivered to the current process. This coexists with the **signal**() and **sigaction**() facilities, and has a lower precedence. The filter will record all attempts to deliver a signal to a process, even if the signal has been marked as SIG_IGN. Event notification happens after normal signal delivery processing. *data* returns the number of times the signal has occurred since the last call to **kevent**(). This filter automatically sets the EV_CLEAR flag internally.

EVFILT_TIMER Establishes an arbitrary timer identified by *ident*. When adding a timer, *data* specifies the timeout period in milliseconds. The timer will be periodic unless EV_ONESHOT is specified. On return, *data* contains the number of times the timeout has expired since the last call to **kevent**(). This filter automatically sets the EV_CLEAR flag internally.

RETURN VALUES

kqueue() creates a new kernel event queue and returns a file descriptor. If there was an error creating the kernel event queue, a value of −1 is returned and errno set.

kevent() returns the number of events placed in the *eventlist*, up to the value given by *nevents*. If an error occurs while processing an element of the *changelist* and there is enough room in the *eventlist*, then the event will be placed in the *eventlist* with `EV_ERROR` set in *flags* and the system error in *data*. Otherwise, `−1` will be returned, and `errno` will be set to indicate the error condition. If the time limit expires, then **kevent**() returns 0.

ERRORS

The **kqueue**() function fails if:

[ENOMEM]	The kernel failed to allocate enough memory for the kernel queue.
[EMFILE]	The per-process descriptor table is full.
[ENFILE]	The system file table is full.

The **kevent**() function fails if:

[EACCES]	The process does not have permission to register a filter.
[EFAULT]	There was an error reading or writing the *kevent* structure.
[EBADF]	The specified descriptor is invalid.
[EINTR]	A signal was delivered before the timeout expired and before any events were placed on the kqueue for return.
[EINVAL]	The specified time limit or filter is invalid.
[ENOENT]	The event could not be found to be modified or deleted.
[ENOMEM]	No memory was available to register the event.
[ESRCH]	The specified process to attach to does not exist.

SEE ALSO

ioctl(2), poll(2), read(2), select(2), sigaction(2), write(2), signal(3), kfilter_register(9), knote(9)

HISTORY

The **kqueue**() and **kevent**() functions first appeared in FreeBSD 4.1, and then in NetBSD 2.0.

AUTHORS

The **kqueue**() system and this manual page were written by Jonathan Lemon ⟨jlemon@FreeBSD.org⟩. NetBSD port and manpage additions were done by
Luke Mewburn ⟨lukem@NetBSD.org⟩,
Jason Thorpe ⟨thorpej@NetBSD.org⟩, and
Jaromir Dolecek ⟨jdolecek@NetBSD.org⟩.

NAME

ktrace — process tracing

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/uio.h>
#include <sys/ktrace.h>

int
ktrace(*const char *tracefile*, *int ops*, *int trpoints*, *pid_t pid*);

int
fktrace(*int fd*, *int ops*, *int trpoints*, *pid_t pid*);

DESCRIPTION

The **ktrace**() function enables or disables tracing of one or more processes. Users may only trace their own processes. Only the super-user can trace setuid or setgid programs.

The *tracefile* gives the pathname of the file to be used for tracing. The file must exist and be writable by the calling process. All trace records are always appended to the file, so the file must be truncated to zero length to discard previous trace data. If tracing points are being disabled (see KTROP_CLEAR below), *tracefile* may be NULL. If using **fktrace**() then instead of passing a filename as *tracefile*, a file descriptor is passed as *fd* and behaviour is otherwise the same.

The **ops** parameter specifies the requested ktrace operation. The defined operations are:

KTROP_SET	Enable trace points specified in *trpoints*.
KTROP_CLEAR	Disable trace points specified in *trpoints*.
KTROP_CLEARFILE	Stop all tracing.
KTRFLAG_DESCEND	The tracing change should apply to the specified process and all its current children.

The **trpoints** parameter specifies the trace points of interest. The defined trace points are:

KTRFAC_SYSCALL	Trace system calls.
KTRFAC_SYSRET	Trace return values from system calls.
KTRFAC_NAMEI	Trace name lookup operations.
KTRFAC_GENIO	Trace all I/O (note that this option can generate much output).
KTRFAC_PSIG	Trace posted signals.
KTRFAC_CSW	Trace context switch points.
KTRFAC_EMUL	Trace emulation changes.
KTRFAC_INHERIT	Inherit tracing to future children.

Each tracing event outputs a record composed of a generic header followed by a trace point specific structure. The generic header is:

```
struct ktr_header {
        int     ktr_len;              /* length of buf */
        short   ktr_type;             /* trace record type */
        pid_t   ktr_pid;              /* process id */
        char    ktr_comm[MAXCOMLEN+1]; /* command name */
        struct  timeval ktr_time;     /* timestamp */
        caddr_t ktr_buf;
};
```

The **ktr_len** field specifies the length of the **ktr_type** data that follows this header. The **ktr_pid** and **ktr_comm** fields specify the process and command generating the record. The **ktr_time** field gives the time (with microsecond resolution) that the record was generated. The **ktr_buf** is an internal kernel pointer and is not useful.

The generic header is followed by **ktr_len** bytes of a **ktr_type** record. The type specific records are defined in the `<sys/ktrace.h>` include file.

RETURN VALUES

On successful completion a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to show the error.

ERRORS

ktrace() will fail if:

[ENOTDIR]	A component of the path prefix is not a directory.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.
[ENOENT]	The named tracefile does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EIO]	An I/O error occurred while reading from or writing to the file system.

SEE ALSO

kdump(1), ktrace(1)

HISTORY

A **ktrace** function call first appeared in 4.4BSD.

NAME

lfs_bmapv — retrieve disk addresses for arrays of blocks

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/types.h>
#include <ufs/lfs/lfs.h>

int
lfs_bmapv(*fsid_t ∗fsidp*, *BLOCK_INFO ∗blkiov*, *int blkcnt*);

DESCRIPTION

lfs_bmapv() fills in the bi_daddr field for every block listed in the block array *blkiov* with the disk address corresponding to the logical block *bi_lbn* of the file with inode *bi_inode*. If *bi_lbn* is LFS_UNUSED_LBN, the disk location of the inode block containing the file's inode will be returned in *bi_daddr* instead.

The *fsidp* argument contains the id of the file system to which the inodes and blocks belong. The *blkiov* argument is an array of BLOCK_INFO structures (see below). The *blkcnt* argument determines the size of the *blkiov* array.

```
typedef struct block_info {
    ino_t       bi_inode;     /∗ inode # ∗/
    ufs_daddr_t bi_lbn;       /∗ logical block w/in file ∗/
    ufs_daddr_t bi_daddr;     /∗ disk address of block ∗/
    time_t      bi_segcreate; /∗ origin segment create time ∗/
    int         bi_version;   /∗ file version number ∗/
    void       ∗bi_bp;        /∗ data buffer ∗/
    int         bi_size;      /∗ size of the block (if fragment) ∗/
} BLOCK_INFO;
```

RETURN VALUES

lfs_bmapv() returns 0 on success, or −1 on error.

ERRORS

An error return from **lfs_bmapv**() indicates:

[EFAULT]	*fsidp* points outside the process's allocated address space.
[EINVAL]	∗*fsidp* does not specify a valid file system.

SEE ALSO

lfs_markv(2), lfs_segclean(2), lfs_segwait(2), lfs_cleanerd(8)

HISTORY

The **lfs_bmapv**() function call appeared in 4.4BSD.

NAME

lfs_markv — rewrite disk blocks to new disk locations

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/types.h>
#include <ufs/lfs/lfs.h>

int
lfs_markv(*fsid_t ∗fsidp*, *BLOCK_INFO ∗blkiov*, *int blkcnt*);

DESCRIPTION

lfs_markv() rewrites the blocks specified in *blkiov* to new disk locations, for the purposes of grouping them next to one another, or to move them out of a segment to clean it. All fields of the BLOCK_INFO structure must be filled in, except for *bi_segcreate*. If *bi_daddr* is not the correct current address for logical block *bi_lbn* of the file with inode number *bi_inode*, or if the file's version number does not match *bi_version*, the block will not be written to disk, but no error will be returned.

The *fsidp* argument contains the id of the filesystem to which the inodes and blocks belong. The *bi_bp* field contains *bi_size* bytes of data to be written into the appropriate block. If *bi_lbn* is specified as LFS_UNUSED_LBN, the inode itself will be rewritten.

The *blkiov* argument is an array of BLOCK_INFO structures (see below). The *blkcnt* argument determines the size of the *blkiov* array.

```
typedef struct block_info {
    ino_t       bi_inode;     /∗ inode # ∗/
    ufs_daddr_t bi_lbn;       /∗ logical block w/in file ∗/
    ufs_daddr_t bi_daddr;     /∗ disk address of block ∗/
    time_t      bi_segcreate; /∗ origin segment create time ∗/
    int         bi_version;   /∗ file version number ∗/
    void       ∗bi_bp;        /∗ data buffer ∗/
    int         bi_size;      /∗ size of the block (if fragment) ∗/
} BLOCK_INFO;
```

RETURN VALUES

lfs_markv() returns 0 on success, or −1 on error.

ERRORS

An error return from **lfs_markv**() indicates:

[EFAULT]	*fsidp* points outside the process's allocated address space.
[EINVAL]	∗*fsidp* does not specify a valid filesystem.
[EBUSY]	One or more of the inodes whose blocks were to be written was locked, and its blocks were not rewritten.

SEE ALSO

lfs_segclean(2), lfs_segwait(2), lfs_cleanerd(8)

HISTORY

The **lfs_markv**() function call appeared in 4.4BSD.

BUGS

The functionality of **lfs_markv**() does not really belong in user space. Among other things it could be used to work around the SF_IMMUTABLE and SF_APPEND file flags (see chflags(2)).

NAME

lfs_segclean — mark a segment clean

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/types.h>

int
lfs_segclean(*fsid_t ∗fsidp*, *u_long segment*);

DESCRIPTION

lfs_segclean() marks segment number *segment* in LFS filesystem ∗*fsidp* "clean" or available for writing.

RETURN VALUES

lfs_segclean() returns 0 on success, or −1 on error.

ERRORS

An error return from **lfs_segclean**() indicates:

[EFAULT]	*fsidp* points outside the process's allocated address space.
[EINVAL]	∗*fsidp* does not specify a valid filesystem.
[EBUSY]	*segment* is marked SU_ACTIVE, meaning that it does not yet belong to a valid checkpoint.

SEE ALSO

lfs_bmapv(2), lfs_markv(2), lfs_segwait(2), lfs_cleanerd(8)

HISTORY

The **lfs_segclean**() function call appeared in 4.4BSD.

NAME

lfs_segwait — wait until a segment is written

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/types.h>

int
lfs_segwait(*fsid_t ∗fsidp*, *struct timeval ∗tv*);

DESCRIPTION

lfs_segwait() blocks until a new segment is acquired for writing by the filesystem specified by ∗*fsidp* or if ∗*fsidp* is −1, until a segment is acquired for writing by any LFS filesystem.

If *timeout* is non-zero, **lfs_segwait**() will return after *timeout* milliseconds regardless of whether a new segment has been designated for writing or not.

RETURN VALUES

lfs_segwait() returns 0 if a new segment was acquired; 1 if it timed out; or −1 on error.

ERRORS

An error return from **lfs_segwait**() indicates:

[EFAULT]	*fsidp* points outside the process's allocated address space.
[EINTR]	A signal was delivered before the time limit expired and before a new segment was designated for writing.
[EINVAL]	The specified time limit is negative.

SEE ALSO

lfs_bmapv(2), lfs_markv(2), lfs_segclean(2), lfs_cleanerd(8)

HISTORY

The **lfs_segwait**() function call appeared in 4.4BSD.

NAME

link — make a hard file link

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
link(*const char ∗name1*, *const char ∗name2*);

DESCRIPTION

The **link**() function call atomically creates the specified directory entry (hard link) *name2* with the attributes of the underlying object pointed at by *name1*. If the link is successful: the link count of the underlying object is incremented; *name1* and *name2* share equal access and rights to the underlying object.

If *name1* is removed, the file *name2* is not deleted and the link count of the underlying object is decremented.

name1 must exist for the hard link to succeed and both *name1* and *name2* must be in the same file system. *name1* may not be a directory unless the caller is the super-user and the file system containing it supports linking to directories.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

link() will fail and no link will be created if:

[ENOTDIR]	A component of either path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	A component of either path prefix does not exist.
[EACCES]	A component of either path prefix denies search permission, or the requested link requires writing in a directory with a mode that denies write permission.
[ELOOP]	Too many symbolic links were encountered in translating one of the pathnames.
[ENOENT]	The file named by *name1* does not exist.
[EOPNOTSUPP]	The file system containing the file named by *name1* does not support links.
[EMLINK]	The link count of the file named by *name1* would exceed {LINK_MAX}.
[EEXIST]	The link named by *name2* does exist.
[EPERM]	The file named by *name1* is a directory and the effective user ID is not super-user, or the file system containing the file does not permit the use of **link**() on a directory.
[EXDEV]	The link named by *name2* and the file named by *name1* are on different file systems.
[ENOSPC]	The directory in which the entry for the new link is being placed cannot be extended

	because there is no space left on the file system containing the directory.
`[EDQUOT]`	The directory in which the entry for the new link is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
`[EIO]`	An I/O error occurred while reading from or writing to the file system to make the directory entry.
`[EROFS]`	The requested link requires writing in a directory on a read-only file system.
`[EFAULT]`	One of the pathnames specified is outside the process's allocated address space.

SEE ALSO

`symlink`(2), `unlink`(2)

STANDARDS

The **`link`**() function conforms to ISO/IEC 9945-1:1990 ("POSIX.1").

NAME

listen — listen for connections on a socket

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/socket.h>

int
listen(*int s*, *int backlog*);

DESCRIPTION

To accept connections, a socket is first created with socket(2), a willingness to accept incoming connections and a queue limit for incoming connections are specified with **listen**(), and then the connections are accepted with accept(2). The **listen**() call applies only to sockets of type SOCK_STREAM or SOCK_SEQPACKET.

The *backlog* parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full the client may receive an error with an indication of ECONNREFUSED, or, if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.

RETURN VALUES

A 0 return value indicates success; −1 indicates an error.

ERRORS

listen() will fail if:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is not a socket.
[EOPNOTSUPP]	The socket is not of a type that supports the operation **listen**().

SEE ALSO

accept(2), connect(2), socket(2)

HISTORY

The **listen**() function call appeared in 4.2BSD.

BUGS

The *backlog* is currently limited (silently) to 128.

NAME

lseek, **seek** — reposition read/write file offset

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

off_t
lseek(*int fildes*, *off_t offset*, *int whence*);

DESCRIPTION

The **lseek**() function repositions the offset of the file descriptor *fildes* to the argument *offset* according to the directive *whence*. The argument *fildes* must be an open file descriptor. **lseek**() repositions the file pointer *fildes* as follows:

If *whence* is SEEK_SET, the offset is set to *offset* bytes.

If *whence* is SEEK_CUR, the offset is set to its current location plus *offset* bytes.

If *whence* is SEEK_END, the offset is set to the size of the file plus *offset* bytes.

The **lseek**() function allows the file offset to be set beyond the end of the existing end-of-file of the file. If data is later written at this point, subsequent reads of the data in the gap return bytes of zeros (until data is actually written into the gap).

Some devices are incapable of seeking. The value of the pointer associated with such a device is undefined.

RETURN VALUES

Upon successful completion, **lseek**() returns the resulting offset location as measured in bytes from the beginning of the file. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

lseek() will fail and the file pointer will remain unchanged if:

[EBADF]	*fildes* is not an open file descriptor.
[ESPIPE]	*fildes* is associated with a pipe, socket, or FIFO.
[EINVAL]	*whence* is not a proper value, or the resulting file offset would be invalid.

SEE ALSO

dup(2), open(2)

STANDARDS

The **lseek**() function conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

BUGS

This document’s use of *whence* is incorrect English, but is maintained for historical reasons.

NAME

m68k_sync_icache — instruction cache synchronization

LIBRARY

m68k Architecture Library (libm68k, −lm68k)

SYNOPSIS

#include <sys/types.h>
#include <m68k/sync_icache.h>

void
m68k_sync_icache(*void *start*, *size_t size*);

DESCRIPTION

m68k_sync_icache() synchronizes data and instruction caches over the specified region. It should be called prior to executing newly generated code.

The affected address range starts at *start* and continues for *size* bytes. If *start* is 0, all the address space of the current execution thread is affected. Addresses outside the specified region may be synchronized, too.

The call always succeeds.

SEE ALSO

arm32_sync_icache(2)

HISTORY

m68k_sync_icache() appeared first in NetBSD 1.4.

AUTHORS

Ignatios Souvatzis

NAME

madvise — give advice about use of memory

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/mman.h>

int
madvise(*void *addr*, *size_t len*, *int behav*);

int
posix_madvise(*void *addr*, *size_t len*, *int advice*);

DESCRIPTION

The **madvise**() system call allows a process that has knowledge of its memory behavior to describe it to the system. The **posix_madvise**() interface is identical and is provided for standards conformance.

The known behaviors are:

`MADV_NORMAL` Tells the system to revert to the default paging behavior.

`MADV_RANDOM` Is a hint that pages will be accessed randomly, and prefetching is likely not advantageous.

`MADV_SEQUENTIAL`
Causes the VM system to depress the priority of pages immediately preceding a given page when it is faulted in.

`MADV_WILLNEED`
Causes pages that are in a given virtual address range to temporarily have higher priority, and if they are in memory, decrease the likelihood of them being freed. Additionally, the pages that are already in memory will be immediately mapped into the process, thereby eliminating unnecessary overhead of going through the entire process of faulting the pages in. This WILL NOT fault pages in from backing store, but quickly map the pages already in memory into the calling process.

`MADV_DONTNEED`
Allows the VM system to decrease the in-memory priority of pages in the specified range. Additionally future references to this address range will incur a page fault.

`MADV_FREE` Gives the VM system the freedom to free pages, and tells the system that information in the specified page range is no longer important.

Portable programs that call the **posix_madvise**() interface should use the aliases `POSIX_MADV_NORMAL`, `POSIX_MADV_SEQUENTIAL`, `POSIX_MADV_RANDOM`, `POSIX_MADV_WILLNEED`, and `POSIX_MADV_DONTNEED` rather than the flags described above.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

madvise() will fail if:

`[EINVAL]` Invalid parameters were provided.

SEE ALSO

`mincore`(2), `mprotect`(2), `msync`(2), `munmap`(2), `posix_fadvise`(2)

STANDARDS

The **`posix_madvise`**() system call is expected to conform to the IEEE Std 1003.1-2001 (“POSIX.1”) standard.

HISTORY

The **`madvise`** system call first appeared in 4.4BSD, but until NetBSD 1.5 it did not perform any of the requests on, or change any behavior of the address range given. The **`posix_madvise`**() was invented in NetBSD 5.0.

NAME

mincore — determine residency of memory pages

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/mman.h>

int
mincore(*void *addr*, *size_t len*, *char *vec*);

DESCRIPTION

The **mincore**() system call allows a process to obtain information about whether pages are core resident. The status of the memory range is returned in the character-per-page array *vec*. If the page is resident, the least significant bit of the corresponding character in *vec* will be set. Other bits are reserved for additional information which future implementations may return.

Note that the status of each page may change between the call to **mincore**() and the return of the page status information. In order to guarantee that pages will remain in core, the address range must be locked with mlock(2).

RETURN VALUES

mincore() returns 0 on success, or −1 on failure and sets the variable *errno* to indicate the error.

ERRORS

The **mincore**() call will fail if:

[EFAULT]	*vec* points to an illegal address.
[EINVAL]	*addr* is not a multiple of the system page size.
[EINVAL]	*len* is equal to 0.
[ENOMEM]	The address range specified is invalid for the calling process, or one or more of the pages specified in the range are not mapped.

SEE ALSO

madvise(2), mlock(2), mprotect(2), msync(2), munmap(2), sysconf(3)

HISTORY

The **mincore**() function first appeared in 4.4BSD.

NAME

minherit — control the inheritance of pages

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/mman.h>

int
minherit(*void *addr*, *size_t len*, *int inherit*);

DESCRIPTION

The **minherit**() system call changes the specified range of virtual addresses to have the specified fork-time inheritance characteristic *inherit*, which can be set to MAP_INHERIT_NONE, MAP_INHERIT_COPY, or MAP_INHERIT_SHARE. Also possible is MAP_INHERIT_DEFAULT, which defaults to MAP_INHERIT_COPY. Not all implementations will guarantee that the inheritance characteristic can be set on a page basis; the granularity of changes may be as large as an entire region.

Normally, the entire address space is marked MAP_INHERIT_COPY; when the process calls **fork**(), the child receives a (virtual) copy of the entire address space. Pages or regions marked MAP_INHERIT_SHARE are shared between the address spaces, while pages or regions marked MAP_INHERIT_NONE will be unmapped in the child.

RETURN VALUES

The **minherit**() function returns the value 0 if successful; otherwise the value −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

minherit() will fail if:

[EINVAL]	An invalid region or invalid parameters were specified.

SEE ALSO

fork(2), madvise(2), mincore(2), mprotect(2), msync(2), munmap(2)

HISTORY

The **minherit**() function first appeared in OpenBSD.

BUGS

If a particular port does not support page-granularity inheritance, there's no way to figure out how large a region is actually affected by **minherit**().

NAME

mkdir — make a directory file

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/stat.h>

int
mkdir(*const char *path*, *mode_t mode*);

DESCRIPTION

The directory *path* is created with the access permissions specified by *mode* and restricted by the umask(2) of the calling process.

The directory's owner ID is set to the process's effective user ID. The directory's group ID is set to that of the parent directory in which it is created.

RETURN VALUES

A 0 return value indicates success. A −1 return value indicates an error, and an error code is stored in *errno*.

ERRORS

mkdir() will fail and no directory will be created if:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	A component of the path prefix does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EROFS]	The named file resides on a read-only file system.
[EEXIST]	The named file exists.
[ENOSPC]	The new directory cannot be created because there is no space left on the file system that will contain the directory.
[ENOSPC]	There are no free inodes on the file system on which the directory is being created.
[EDQUOT]	The new directory cannot be created because the user's quota of disk blocks on the file system that will contain the directory has been exhausted.
[EDQUOT]	The user's quota of inodes on the file system on which the directory is being created has been exhausted.
[EIO]	An I/O error occurred while making the directory entry or allocating the inode.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[EFAULT]	*path* points outside the process's allocated address space.

SEE ALSO

`chmod`(2), `stat`(2), `umask`(2)

STANDARDS

The **`mkdir`**() function conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

NAME

mkfifo — make a fifo file

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/stat.h>

int
mkfifo(*const char *path*, *mode_t mode*);

DESCRIPTION

mkfifo() creates a new fifo file with name *path*. The access permissions are specified by *mode* and restricted by the umask(2) of the calling process.

The fifo's owner ID is set to the process's effective user ID. The fifo's group ID is set to that of the parent directory in which it is created.

RETURN VALUES

A 0 return value indicates success. A −1 return value indicates an error, and an error code is stored in *errno*.

ERRORS

mkfifo() will fail and no fifo will be created if:

[EOPNOTSUPP]	The kernel has not been configured to support fifo's.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	A component of the path prefix does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EROFS]	The named file resides on a read-only file system.
[EEXIST]	The named file exists.
[ENOSPC]	The directory in which the entry for the new fifo is being placed cannot be extended because there is no space left on the file system containing the directory.
[ENOSPC]	There are no free inodes on the file system on which the fifo is being created.
[EDQUOT]	The directory in which the entry for the new fifo is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
[EDQUOT]	The user's quota of inodes on the file system on which the fifo is being created has been exhausted.
[EIO]	An I/O error occurred while making the directory entry or allocating the inode.
[EIO]	An I/O error occurred while reading from or writing to the file system.

[EFAULT] *path* points outside the process's allocated address space.

SEE ALSO

chmod(2), stat(2), umask(2)

STANDARDS

The **mkfifo** function call conforms to ISO/IEC 9945-1:1990 ("POSIX.1").

NAME

mknod — make a special file node

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/stat.h>

int
mknod(*const char *path*, *mode_t mode*, *dev_t dev*);

DESCRIPTION

The device special file *path* is created with the major and minor device numbers specified by *dev*. The access permissions of *path* are extracted from *mode*, modified by the umask(2) of the parent process.

mknod() requires super-user privileges.

RETURN VALUES

Upon successful completion a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

mknod() will fail and the file will be not created if:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	A component of the path prefix does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EPERM]	The process's effective user ID is not super-user.
[EIO]	An I/O error occurred while making the directory entry or allocating the inode.
[ENOSPC]	The directory in which the entry for the new node is being placed cannot be extended because there is no space left on the file system containing the directory.
[ENOSPC]	There are no free inodes on the file system on which the node is being created.
[EDQUOT]	The directory in which the entry for the new node is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
[EDQUOT]	The user's quota of inodes on the file system on which the node is being created has been exhausted.
[EROFS]	The named file resides on a read-only file system.
[EEXIST]	The named file exists.
[EFAULT]	*path* points outside the process's allocated address space.

SEE ALSO

`chmod`(2), `mkfifo`(2), `stat`(2), `umask`(2)

HISTORY

A **`mknod`**() function call appeared in Version 6 AT&T UNIX.

NAME

mlock, **munlock** — lock (unlock) physical pages in memory

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/mman.h>

int
mlock(*void *addr*, *size_t len*);

int
munlock(*void *addr*, *size_t len*);

DESCRIPTION

The **mlock** system call locks into memory the physical pages associated with the virtual address range starting at *addr* for *len* bytes. The **munlock** call unlocks pages previously locked by one or more **mlock** calls. For both, the *addr* parameter should be aligned to a multiple of the page size. If the *len* parameter is not a multiple of the page size, it will be rounded up to be so. The entire range must be allocated.

After an **mlock** call, the indicated pages will cause neither a non-resident page nor address-translation fault until they are unlocked. They may still cause protection-violation faults or TLB-miss faults on architectures with software-managed TLBs. The physical pages remain in memory until all locked mappings for the pages are removed. Multiple processes may have the same physical pages locked via their own virtual address mappings. A single process may likewise have pages multiply-locked via different virtual mappings of the same pages or via nested **mlock** calls on the same address range. Unlocking is performed explicitly by **munlock** or implicitly by a call to **munmap** which deallocates the unmapped address range. Locked mappings are not inherited by the child process after a fork(2).

Since physical memory is a potentially scarce resource, processes are limited in how much they can lock down. A single process can **mlock** the minimum of a system-wide ‘‘wired pages’’ limit and the per-process RLIMIT_MEMLOCK resource limit.

RETURN VALUES

A return value of 0 indicates that the call succeeded and all pages in the range have either been locked or unlocked. A return value of −1 indicates an error occurred and the locked status of all pages in the range remains unchanged. In this case, the global location *errno* is set to indicate the error.

ERRORS

mlock() will fail if:

[EINVAL]	The address given is not page aligned or the length is negative.
[EAGAIN]	Locking the indicated range would exceed either the system or per-process limit for locked memory.
[ENOMEM]	Some portion of the indicated address range is not allocated. There was an error faulting/mapping a page.
[EPERM]	**mlock**() was called by non-root on an architecture where locked page accounting is not implemented.

munlock() will fail if:

`[EINVAL]`	The address given is not page aligned or the length is negative.
`[ENOMEM]`	Some portion of the indicated address range is not allocated. Some portion of the indicated address range is not locked.

SEE ALSO

`fork`(2), `mincore`(2), `mmap`(2), `munmap`(2), `setrlimit`(2), `getpagesize`(3)

STANDARDS

The **`mlock`**() and **`munlock`**() functions conform to IEEE Std 1003.1b-1993 (“POSIX.1”).

HISTORY

The **`mlock`**() and **`munlock`**() functions first appeared in 4.4BSD.

BUGS

The per-process resource limit is a limit on the amount of virtual memory locked, while the system-wide limit is for the number of locked physical pages. Hence a process with two distinct locked mappings of the same physical page counts as 2 pages against the per-process limit and as only a single page in the system limit.

NAME

mlockall, **munlockall** — lock (unlock) the address space of a process

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/mman.h>

int
mlockall(*int flags*);

int
munlockall(*void*);

DESCRIPTION

The **mlockall** system call locks into memory the physical pages associated with the address space of a process until the address space is unlocked, the process exits, or execs another program image.

The following flags affect the behavior of **mlockall**:

MCL_CURRENT	Lock all pages currently mapped into the process's address space.
MCL_FUTURE	Lock all pages mapped into the process's address space in the future, at the time the mapping is established. Note that this may cause future mappings to fail if those mappings cause resource limits to be exceeded.

Since physical memory is a potentially scarce resource, processes are limited in how much they can lock down. A single process can lock the minimum of a system-wide "wired pages" limit and the per-process RLIMIT_MEMLOCK resource limit.

The **munlockall** call unlocks any locked memory regions in the process address space. Any regions mapped after an **munlockall** call will not be locked.

RETURN VALUES

A return value of 0 indicates that the call succeeded and all pages in the range have either been locked or unlocked. A return value of −1 indicates an error occurred and the locked status of all pages in the range remains unchanged. In this case, the global location *errno* is set to indicate the error.

ERRORS

mlockall() will fail if:

[EINVAL]	The *flags* argument is zero, or includes unimplemented flags.
[ENOMEM]	Locking the indicated range would exceed either the system or per-process limit for locked memory.
[EAGAIN]	Some or all of the memory mapped into the process's address space could not be locked when the call was made.
[EPERM]	The calling process does not have the appropriate privilege to perform the requested operation.

SEE ALSO

mincore(2), mlock(2), mmap(2), munmap(2), setrlimit(2)

STANDARDS

The **mlockall**() and **munlockall**() functions conform to IEEE Std 1003.1b-1993 (“POSIX.1”).

HISTORY

The **mlockall**() and **munlockall**() functions first appeared in NetBSD 1.5.

BUGS

The per-process resource limit is a limit on the amount of virtual memory locked, while the system-wide limit is for the number of locked physical pages. Hence a process with two distinct locked mappings of the same physical page counts as 2 pages against the per-process limit and as only a single page in the system limit.

NAME

mmap — map files or devices into memory

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/mman.h>

void ∗
mmap(*void ∗addr*, *size_t len*, *int prot*, *int flags*, *int fd*, *off_t offset*);

DESCRIPTION

The **mmap** function causes the pages starting at *addr* and continuing for at most *len* bytes to be mapped from the object described by *fd*, starting at byte offset *offset*. If *len* is not a multiple of the pagesize, the mapped region may extend past the specified range. Any such extension beyond the end of the mapped object will be zero-filled.

If *addr* is non-zero, it is used as a hint to the system. (As a convenience to the system, the actual address of the region may differ from the address supplied.) If *addr* is zero, an address will be selected by the system. The actual starting address of the region is returned. A successful *mmap* deletes any previous mapping in the allocated address range.

The protections (region accessibility) are specified in the *prot* argument by *OR*'ing the following values:

PROT_EXEC	Pages may be executed.
PROT_READ	Pages may be read.
PROT_WRITE	Pages may be written.
PROT_NONE	Pages may not be accessed.

Note that, due to hardware limitations, on some platforms PROT_WRITE **may imply** PROT_READ**, and** PROT_READ **may imply** PROT_EXEC**. Portable programs should not rely on these flags being separately enforceable.**

The *flags* parameter specifies the type of the mapped object, mapping options and whether modifications made to the mapped copy of the page are private to the process or are to be shared with other references. Note that either MAP_SHARED, MAP_PRIVATE or MAP_COPY must be specified. Sharing, mapping type and options are specified in the *flags* argument by *OR*'ing the following values:

MAP_ALIGNED(n)	Request that the allocation be aligned to the given boundary. The parameter *n* should be the base 2 logarithm of the desired alignment (e.g., to request alignment to 16K, use 14 as the value for n). The alignment must be equal to or greater than the platform's page size as returned by sysconf(3) with the _SC_PAGESIZE request.
MAP_ANON	Map anonymous memory not associated with any specific file. The file descriptor is not used for creating MAP_ANON regions, and must be specified as −1. The mapped memory will be zero filled.
MAP_FILE	Mapped from a regular file or character-special device memory.
MAP_FIXED	Do not permit the system to select a different address than the one specified. If the specified address cannot be used, **mmap** will fail. If MAP_FIXED is specified, *addr* must be a multiple of the pagesize. Use of this option is discouraged.

`MAP_HASSEMAPHORE`	Notify the kernel that the region may contain semaphores and that special handling may be necessary.
`MAP_INHERIT`	Permit regions to be inherited across `execve`(2) system calls.
`MAP_TRYFIXED`	Attempt to use the address *addr* even if it falls within the normally protected process data or text segment memory regions. If the requested region of memory is actually present in the memory map, a different address will be selected as if MAP_TRYFIXED had not been specified. If *addr* is *NULL*, this flag is ignored and the system will select a mapping address.
`MAP_WIRED`	Lock the mapped region into memory as with `mlock`(2).
`MAP_PRIVATE`	Modifications made by this process are private, however modifications made by other processes using `MAP_SHARED` will be seen.
`MAP_SHARED`	Modifications are shared.
`MAP_COPY`	Modifications are private, including other processes.

The `close`(2) function does not unmap pages, see `munmap`(2) for further information.

The current design does not allow a process to specify the location of swap space. In the future we may define an additional mapping type, `MAP_SWAP`, in which the file descriptor argument specifies a file or device to which swapping should be done.

If `MAP_FIXED` is not specified, the system will attempt to place the mapping in an unused portion of the address space chosen to minimize possible collision between mapped regions and the heap.

RETURN VALUES

Upon successful completion, **mmap** returns a pointer to the mapped region. Otherwise, a value of `MAP_FAILED` is returned and *errno* is set to indicate the error. The symbol `MAP_FAILED` is defined in the header ⟨`sys/mman.h`⟩. No successful return from **mmap**() will return the value `MAP_FAILED`.

ERRORS

mmap() will fail if:

[EACCES]	The flag `PROT_READ` was specified as part of the *prot* parameter and *fd* was not open for reading. The flags `MAP_SHARED` and `PROT_WRITE` were specified as part of the *flags* and *prot* parameters and *fd* was not open for writing.
[EBADF]	*fd* is not a valid open file descriptor.
[EINVAL]	`MAP_FIXED` was specified and the *addr* parameter was not page aligned or was outside of the valid address range for a process. `MAP_ANON was specified and` *fd* was not −1.
[ENODEV]	*fd* did not reference a regular or character special file.
[ENOMEM]	`MAP_FIXED` was specified and the *addr* parameter wasn't available. `MAP_ANON` was specified and insufficient memory was available.
[EOVERFLOW]	*fd* references a regular file and the value of *offset* plus *len* would exceed the offset maximum established in its open file description.

SEE ALSO

`madvise`(2), `mincore`(2), `mlock`(2), `mprotect`(2), `msync`(2), `munmap`(2), `getpagesize`(3), `sysconf`(3)

BUGS

The `MAP_COPY` flag is not implemented. The current `MAP_COPY` semantics are the same as those of the `MAP_PRIVATE` flag.

NAME

mount, **unmount** — mount or dismount a file system

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/mount.h>

int
mount(*const char *type*, *const char *dir*, *int flags*, *void *data*,
    size_t data_len);

int
unmount(*const char *dir*, *int flags*);

DESCRIPTION

The **mount**() function grafts a file system object onto the system file tree at the point *dir*. The argument *data* describes the file system object to be mounted, and is *data_len* bytes long. The argument *type* tells the kernel how to interpret *data* (See *type* below). The contents of the file system become available through the new mount point *dir*. Any files in *dir* at the time of a successful mount are swept under the carpet so to speak, and are unavailable until the file system is unmounted.

The following *flags* may be specified to suppress default semantics which affect file system access.

MNT_RDONLY	The file system should be treated as read-only; even the super-user may not write on it.
MNT_NOEXEC	Do not allow files to be executed from the file system.
MNT_NOSUID	Do not honor setuid or setgid bits on files when executing them.
MNT_NODEV	Do not interpret special files on the file system.
MNT_UNION	Union with underlying filesystem instead of obscuring it.
MNT_SYNCHRONOUS	All I/O to the file system should be done synchronously.
MNT_ASYNC	All I/O to the file system should be done asynchronously.
MNT_NOCOREDUMP	Do not allow programs to dump core files on the file system.
MNT_NOATIME	Never update access time in the file system.
MNT_SYMPERM	Recognize the permission of symbolic link when reading or traversing.
MNT_NODEVMTIME	Never update modification time of device files.
MNT_SOFTDEP	Use soft dependencies.

The MNT_UPDATE and the MNT_GETARGS flags indicate that the mount command is being applied to an already mounted file system. The MNT_UPDATE flag allows the mount flags to be changed without requiring that the file system be unmounted and remounted. Some file systems may not allow all flags to be changed. For example, most file systems will not allow a change from read-write to read-only. The MNT_GETARGS flag does not alter any of the mounted filesystem's properties, but returns the filesystem-specific arguments for the currently mounted filesystem.

The *type* argument defines the type of the file system. The types of file systems known to the system are defined in ⟨sys/mount.h⟩. *data* is a pointer to a structure that contains the type specific arguments to mount. Some of the currently supported types of file systems and their type specific data are:

```
MOUNT_FFS
     struct ufs_args {
           char      *fspec;             /* block special file to mount */
     };

MOUNT_NFS
     struct nfs_args {
           int              version;      /* args structure version */
           struct sockaddr *addr;         /* file server address */
           int              addrlen;      /* length of address */
           int              sotype;       /* Socket type */
           int              proto;        /* and Protocol */
           u_char          *fh;           /* File handle to be mounted */
           int              fhsize;       /* Size, in bytes, of fh */
           int              flags;        /* flags */
           int              wsize;        /* write size in bytes */
           int              rsize;        /* read size in bytes */
           int              readdirsize;  /* readdir size in bytes */
           int              timeo;        /* initial timeout in .1 secs */
           int              retrans;      /* times to retry send */
           int              maxgrouplist; /* Max. size of group list */
           int              readahead;    /* # of blocks to readahead */
           int              leaseterm;    /* Term (sec) of lease */
           int              deadthresh;   /* Retrans threshold */
           char            *hostname;     /* server's name */
     };

MOUNT_MFS
     struct mfs_args {
           char     *fspec;             /* name to export for statfs */
           struct   export_args30 pad;  /* unused */
           caddr_t  base;               /* base of file system in mem */
           u_long   size;               /* size of file system */
     };
```

The **unmount**() function call disassociates the file system from the specified mount point *dir*.

The *flags* argument may specify `MNT_FORCE` to specify that the file system should be forcibly unmounted even if files are still active. Active special devices continue to work, but any further accesses to any other active files result in errors even if the file system is later remounted.

RETURN VALUES

mount() returns the value 0 if the mount was successful, the number of bytes written to *data* for `MNT_GETARGS`, otherwise −1 is returned and the variable *errno* is set to indicate the error.

unmount() returns the value 0 if the unmount succeeded; otherwise −1 is returned and the variable *errno* is set to indicate the error.

ERRORS

mount() will fail when one of the following occurs:

`[EPERM]`	The caller is not the super-user.
`[ENAMETOOLONG]`	A component of a pathname exceeded `{NAME_MAX}` characters, or an entire path name exceeded `{PATH_MAX}` characters.
`[ELOOP]`	Too many symbolic links were encountered in translating a pathname.
`[ENOENT]`	A component of *dir* does not exist.
`[ENOTDIR]`	A component of *name* is not a directory, or a path prefix of *special* is not a directory.
`[EBUSY]`	Another process currently holds a reference to *dir*.
`[EFAULT]`	*dir* points outside the process's allocated address space.

The following errors can occur for a *ufs* file system mount:

`[ENODEV]`	A component of ufs_args *fspec* does not exist.
`[ENOTBLK]`	*Fspec* is not a block device.
`[ENXIO]`	The major device number of *fspec* is out of range (this indicates no device driver exists for the associated hardware).
`[EBUSY]`	*Fspec* is already mounted.
`[EMFILE]`	No space remains in the mount table.
`[EINVAL]`	The super block for the file system had a bad magic number or an out of range block size.
`[ENOMEM]`	Not enough memory was available to read the cylinder group information for the file system.
`[EIO]`	An I/O error occurred while reading the super block or cylinder group information.
`[EFAULT]`	*Fspec* points outside the process's allocated address space.

The following errors can occur for a *nfs* file system mount:

`[ETIMEDOUT]`	*Nfs* timed out trying to contact the server.
`[EFAULT]`	Some part of the information described by nfs_args points outside the process's allocated address space.

The following errors can occur for a *mfs* file system mount:

`[EMFILE]`	No space remains in the mount table.
`[EINVAL]`	The super block for the file system had a bad magic number or an out of range block size.
`[ENOMEM]`	Not enough memory was available to read the cylinder group information for the file system.
`[EIO]`	A paging error occurred while reading the super block or cylinder group information.
`[EFAULT]`	*Name* points outside the process's allocated address space.

unmount() may fail with one of the following errors:

`[EPERM]`	The caller is not the super-user.

[ENOTDIR]	A component of the path is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EINVAL]	The requested directory is not in the mount table.
[EBUSY]	A process is holding a reference to a file located on the file system.
[EIO]	An I/O error occurred while writing cached file system information.
[EFAULT]	*dir* points outside the process's allocated address space.

A *ufs* or *mfs* mount can also fail if the maximum number of file systems are currently mounted.

SEE ALSO

getvfsstat(2), nfssvc(2), getmntinfo(3), symlink(7), mount(8), sysctl(8), umount(8)

HISTORY

The **mount**() and **umount**() (now **unmount**()) function calls appeared in Version 6 AT&T UNIX.

Prior to NetBSD 4.0 the **mount** call was used to NFS export filesystems. This is now done through **nfssvc**().

The data_len argument was added for NetBSD 5.0.

BUGS

Some of the error codes need translation to more obvious messages.

Far more filesystems are supported than those those listed.

NAME

`mprotect` — control the protection of pages

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

`#include <sys/mman.h>`

`int`
`mprotect`(*`void *addr`*, *`size_t len`*, *`int prot`*);

DESCRIPTION

The **`mprotect`**() system call changes the specified pages to have protection *`prot`*. Not all implementations will guarantee protection on a page basis; the granularity of protection changes may be as large as an entire region.

The protections (region accessibility) are specified in the *`prot`* argument by OR'ing the following values:

`PROT_EXEC`	Pages may be executed.
`PROT_READ`	Pages may be read.
`PROT_WRITE`	Pages may be written.
`PROT_NONE`	No permissions.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

`[EACCES]`	A memory protection violation occurred, or the `PROT_EXECUTE` flag was attempted on pages which belong to a filesystem mounted with the `NOEXEC flag.`
`[EINVAL]`	An invalid memory range, or invalid parameters were provided.
`[ENOMEM]`	A resource shortage occurred while internally calling **`uvm_map_protect`**().

SEE ALSO

`madvise`(2), `mincore`(2), `msync`(2), `munmap`(2)

HISTORY

The **`mprotect`**() function first appeared in 4.4BSD.

NAME

mremap — re-map a virtual memory address

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/mman.h>

void ∗
mremap(*void ∗oldp*, *size_t oldsize*, *void ∗newp*, *size_t newsize*, *int flags*);

DESCRIPTION

The **mremap**() function resizes the mapped range (see mmap(2)) starting at *oldp* and having size *oldsize* to *newsize*. The following arguments can be OR'ed together in the *flags* argument:

MAP_ALIGNED(*n*) The allocation should be aligned to the given boundary, i.e. ensure that the lowest *n* bits of the address are zero. The parameter *n* should be the base 2 logarithm of the desired alignment (e.g., to request alignment to 16K, use 14 as the value for n). The alignment must be equal to or greater than the platform's page size as returned by sysconf(3) with the _SC_PAGESIZE request.

MAP_FIXED *newp* is tried and **mremap**() fails if that address can't be used as new base address for the range. Otherwise, *oldp* and *newp* are used as hints for the position, factoring in the given alignment.

RETURN VALUES

mremap() returns the new address or MAP_FAILED, if the remap failed.

HISTORY

The **mremap**() system call appeared in NetBSD 5.0. It was based on the code that supports **mremap**() compatibility for Linux binaries.

COMPATIBILITY

The semantics of **mremap**() differ from the one provided by glibc on Linux in that the *newp* argument was added and a different set of *flags* are implemented.

SEE ALSO

mmap(2), munmap(2)

NAME

msgctl — message control operations

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/msg.h>

int
msgctl(*int msqid*, *int cmd*, *struct msqid_ds *buf*);

DESCRIPTION

The **msgctl**() system call performs control operations on the message queue specified by *msqid*.

Each message queue has a **msqid_ds** structure associated with it which contains the following members:

```
struct ipc_perm msg_perm; /* msg queue permission bits */
msgqnum_t msg_qnum;       /* # of msgs in the queue */
msglen_t  msg_qbytes;     /* max # of bytes on the queue */
pid_t     msg_lspid;      /* pid of last msgsnd() */
pid_t     msg_lrpid;      /* pid of last msgrcv() */
time_t    msg_stime;      /* time of last msgsnd() */
time_t    msg_rtime;      /* time of last msgrcv() */
time_t    msg_ctime;      /* time of last msgctl() */
```

The **ipc_perm** structure used inside the **msgid_ds** structure is defined in ⟨sys/ipc.h⟩ and contains the following members:

```
uid_t cuid;  /* creator user id */
gid_t cgid;  /* creator group id */
uid_t uid;   /* user id */
gid_t gid;   /* group id */
mode_t mode; /* permission (lower 9 bits) */
```

The operation to be performed by **msgctl**() is specified in *cmd* and is one of:

IPC_STAT Gather information about the message queue and place it in the structure pointed to by *buf*.

IPC_SET Set the value of the *msg_perm.uid*, *msg_perm.gid*, *msg_perm.mode* and *msg_qbytes* fields in the structure associated with *msqid*. The values are taken from the corresponding fields in the structure pointed to by *buf*. This operation can only be executed by the super-user, or a process that has an effective user id equal to either *msg_perm.cuid* or *msg_perm.uid* in the data structure associated with the message queue. The value of *msg_qbytes* can only be increased by the super-user. Values for *msg_qbytes* that exceed the system limit (MSGMNB from ⟨sys/msg.h⟩) are silently truncated to that limit.

IPC_RMID Remove the message queue specified by *msqid* and destroy the data associated with it. Only the super-user or a process with an effective uid equal to the *msg_perm.cuid* or *msg_perm.uid* values in the data structure associated with the queue can do this.

The permission to read from or write to a message queue (see msgsnd(2) and msgrcv(2)) is determined by the *msg_perm.mode* field in the same way as is done with files (see chmod(2)), but the effective uid can match either the *msg_perm.cuid* field or the *msg_perm.uid* field, and the effective gid can match either *msg_perm.cgid* or *msg_perm.gid*.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

msgctl() will fail if:

[EPERM]	*cmd* is equal to IPC_SET or IPC_RMID and the caller is not the super-user, nor does the effective uid match either the *msg_perm.uid* or *msg_perm.cuid* fields of the data structure associated with the message queue.
	An attempt was made to increase the value of *msg_qbytes* through IPC_SET, but the caller is not the super-user.
[EACCES]	*cmd* is IPC_STAT and the caller has no read permission for this message queue.
[EINVAL]	*msqid* is not a valid message queue identifier.
	cmd is not a valid command.
[EFAULT]	*buf* specifies an invalid address.

SEE ALSO

msgget(2), msgrcv(2), msgsnd(2)

STANDARDS

The **msgctl** system call conforms to X/Open System Interfaces and Headers Issue 5 (“XSH5”).

HISTORY

Message queues appeared in the first release of AT&T System V UNIX.

NAME

msgget — get message queue identifier

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/msg.h>

int
msgget(*key_t key*, *int msgflg*);

DESCRIPTION

The **msgget**() system call returns the message queue identifier associated with *key*. A message queue identifier is a unique integer greater than zero.

A message queue is created if either *key* is equal to `IPC_PRIVATE`, or *key* does not have a message queue identifier associated with it and the `IPC_CREAT` bit is set in *msgflg*. If both the `IPC_CREAT` bit and the `IPC_EXCL` bit are set in *msgflg*, and *key* has a message queue identifier associated with it already, the operation will fail.

If a new message queue is created, the data structure associated with it (the *msqid_ds* structure, see `msgctl`(2)) is initialized as follows:

- *msg_perm.cuid* and *msg_perm.uid* are set to the effective uid of the calling process.
- *msg_perm.gid* and *msg_perm.cgid* are set to the effective gid of the calling process.
- *msg_perm.mode* is set to the lower 9 bits of *msgflg*.
- *msg_qnum*, *msg_lspid*, *msg_lrpid*, *msg_rtime*, and *msg_stime* are set to 0.
- *msg_qbytes* is set to the system wide maximum value for the number of bytes in a queue (`MSGMNB`).
- *msg_ctime* is set to the current time.

RETURN VALUES

Upon successful completion a positive message queue identifier is returned. Otherwise, −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EACCES]	A message queue is already associated with *key* and the caller has no permission to access it.
[EEXIST]	Both `IPC_CREAT` and `IPC_EXCL` are set in *msgflg*, and a message queue is already associated with *key*.
[ENOSPC]	A new message queue could not be created because the system limit for the number of message queues has been reached.
[ENOENT]	`IPC_CREAT` is not set in *msgflg* and no message queue associated with *key* was found.

SEE ALSO

`msgctl`(2), `msgrcv`(2), `msgsnd`(2), `ftok`(3)

STANDARDS

The **msgget** system call conforms to X/Open System Interfaces and Headers Issue 5 (“XSH5”).

HISTORY

Message queues appeared in the first release of AT&T System V UNIX.

NAME

msgrcv — receive a message from a message queue

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/msg.h>

ssize_t
msgrcv(*int msqid*, *void *msgp*, *size_t msgsz*, *long msgtyp*, *int msgflg*);

DESCRIPTION

The **msgrcv**() function receives a message from the message queue specified in *msqid*, and places it into the user-defined structure pointed to by *msgp*. This structure must contain a first field of type **long** that will indicate the user-defined type of the message. The remaining fields will contain the contents of the message. The following is an example of what this user-defined structure might look like:

```
struct mymsg {
    long mtype;    /* message type */
    char mtext[1]; /* body of message */
};
```

mtype is an integer greater than 0 that can be used to select messages. *mtext* is an array of bytes, with size up to the system limit MSGMAX.

The value of *msgtyp* has one of the following meanings:

- *msgtyp* is greater than 0. The first message of type *msgtyp* will be received.
- *msgtyp* is equal to 0. The first message on the queue will be received.
- *msgtyp* is less than 0. The first message of the lowest message type that is less than or equal to the absolute value of *msgtyp* will be received.

msgsz specifies the maximum length of the requested message. If the received message has a length greater than *msgsz* it will be silently truncated if the MSG_NOERROR flag is set in *msgflg*, otherwise an error will be returned.

If no matching message is present on the message queue specified by *msqid*, the behaviour of **msgrcv**() depends on whether the IPC_NOWAIT flag is set in *msgflg* or not. If IPC_NOWAIT is set, then **msgrcv**() will immediately return a value of −1 and set *errno* to EAGAIN. If IPC_NOWAIT is not set, the calling process will block until:

- A message of the requested type becomes available on the message queue.
- The message queue is removed, in which case −1 will be returned and *errno* set to EIDRM.
- A signal is received and caught. −1 is returned and *errno* is set to EINTR.

If a message is successfully received, the data structure associated with *msqid* is updated as follows:

- *msg_lrpid* is set to the pid of the caller.
- *msg_lrtime* is set to the current time.
- *msg_qnum* is decremented by 1.

RETURN VALUES

Upon successful completion, **msgrcv**() returns the number of bytes received into the *mtext* field of the structure pointed to by *msgp*. Otherwise, −1 is returned, and *errno* set to indicate the error.

ERRORS

msgrcv() will fail if:

[EINVAL]	*msqid* is not a valid message queue identifier
	The message queue was removed while **msgrcv**() was waiting for a message of the requested type to become available in it.
	msgsz is less than 0.
[E2BIG]	A matching message was received, but its size was greater than *msgsz* and the MSG_NOERROR flag was not set in *msgflg*.
[EACCES]	The calling process does not have read access to the message queue.
[EFAULT]	*msgp* points to an invalid address.
[EINTR]	The system call was interrupted by the delivery of a signal.
[EAGAIN]	There is no message of the requested type available on the message queue, and IPC_NOWAIT is set in *msgflg*.
[EIDRM]	The message queue identifier *msqid* is removed from the system.
[ENOMSG]	The queue does not contain a message of the desired type and IPC_NOWAIT is set.

SEE ALSO

msgctl(2), msgget(2), msgsnd(2)

STANDARDS

The **msgrcv** system call conforms to X/Open System Interfaces and Headers Issue 5 (“XSH5”).

HISTORY

Message queues appeared in the first release of AT&T System V UNIX.

NAME

msgsnd — send a message to a message queue

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/msg.h>

int
msgsnd(*int msqid*, *const void *msgp*, *size_t msgsz*, *int msgflg*);

DESCRIPTION

The **msgsnd**() function sends a message from the message queue specified in *msqid*. *msgp* points to a user-defined structure containing the message. This structure must contain a first field of type **long** that will indicate the user-defined type of the message. The remaining fields will contain the contents of the message. The following is an example of what this user-defined structure might look like:

```
struct mymsg {
    long mtype;    /* message type */
    char mtext[1]; /* body of message */
};
```

mtype is an integer greater than 0 that can be used for selecting messages (see msgrcv(2)). *mtext* is an array of bytes, with size up to the system limit MSGMAX.

If the number of bytes already on the message queue plus *msgsz* is greater than the maximum number of bytes in the message queue (*msg_qbytes*, see msgctl(2)), or if the number of messages on all queues system-wide is already equal to the system limit, *msgflg* determines the action of **msgsnd**(). If *msgflg* has IPC_NOWAIT mask set in it, the call will return immediately. If *msgflg* does not have IPC_NOWAIT set in it, the call will block until:

- The condition which caused the call to block no longer exists. The message was sent.
- The message queue is removed, in which case −1 will be returned and *errno* set to EINVAL.
- The caller catches a signal. The call returns with *errno* set to EINTR.

After a successful call, the data structure associated with the message queue is updated in the following way:

- *msg_qnum* is incremented by 1.
- *msg_lspid* is set to the pid of the calling process.
- *msg_stime* is set to the current time.

RETURN VALUES

Upon successful completion, 0 is returned. Otherwise, −1 is returned and *errno* is set to indicate the error.

ERRORS

msgsnd() will fail if:

[EINVAL]	*msqid* is not a valid message queue identifier, or the value of *mtype* is less than 1.
	The message queue was removed while **msgsnd**() was waiting for a resource to become available in order to deliver the message.

	msgsz is less than 0, or greater than *msg_qbytes*.
[EACCES]	The calling process does not have write access to the message queue.
[EAGAIN]	There was no space for this message either on the queue or in the whole system, and IPC_NOWAIT was set in *msgflg*.
[EFAULT]	*msgp* points to an invalid address.
[EINTR]	The system call was interrupted by the delivery of a signal.

SEE ALSO

msgctl(2), msgget(2), msgrcv(2)

STANDARDS

The **msgsnd** system call conforms to X/Open System Interfaces and Headers Issue 5 (“XSH5”).

HISTORY

Message queues appeared in the first release of AT&T System V UNIX.

NAME

msync — synchronize a mapped region

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/mman.h>

int
msync(*void *addr*, *size_t len*, *int flags*);

DESCRIPTION

The **msync**() system call writes all pages with shared modifications in the specified region of the process's address space back to permanent storage, and, if requested, invalidates cached data mapped in the region. If *len* is 0, all modified pages within the region containing *addr* will be flushed; if *len* is non-zero, only modified pages containing *addr* and *len* succeeding locations will be flushed. Any required synchronization of memory caches will also take place at this time. Filesystem operations on a file that is mapped for shared modifications are unpredictable except after an **msync**().

The *flags* argument is formed by *or*'ing the following values

MS_ASYNC	Perform asynchronous writes.
MS_SYNC	Perform synchronous writes.
MS_INVALIDATE	Invalidate cached data after writing.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

The following errors may be reported:

[EBUSY]	The MS_INVALIDATE flag was specified and a portion of the specified region was locked with mlock(2).
[EINVAL]	The specified *flags* argument was invalid.
[EINVAL]	The *addr* parameter was not page aligned.
[EINVAL]	The *addr* parameter did not specify an address part of a mapped region.
[EINVAL]	The *len* parameter was negative.
[EIO]	An I/O error occurred while writing to the file system.
[ENOMEM]	Addresses in the specified region are outside the range allowed for the address space of the process, or specify one or more pages which are unmapped.

SEE ALSO

mlock(2), mmap(2), munlock(2)

HISTORY

The **msync**() function first appeared in 4.4BSD. It was modified to conform to IEEE Std 1003.1b-1993 ("POSIX.1") in NetBSD 1.3.

NAME

munmap — remove a mapping

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/mman.h>

int
munmap(*void *addr*, *size_t len*);

DESCRIPTION

The **munmap**() system call deletes the mappings for the specified address range, and causes further references to addresses within the range to generate invalid memory references.

RETURN VALUES

Upon successful completion, **munmap** returns zero. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

munmap() will fail if:

[EINVAL]	The *addr* parameter was not page aligned, the *len* parameter was negative, or some part of the region being unmapped is outside the valid address range for a process.

SEE ALSO

madvise(2), mincore(2), mlock(2), mmap(2), mprotect(2), msync(2), getpagesize(3)

HISTORY

The **munmap**() function first appeared in 4.4BSD.

NAME

nanosleep — high resolution sleep

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <time.h>

int
nanosleep(*const struct timespec *rqtp*, *struct timespec *rmtp*);

DESCRIPTION

The **nanosleep**() suspends execution of the calling process until either the number of seconds and nanoseconds specified by *rqtp* have elapsed or a signal is delivered to the calling process and its action is to invoke a signal catching function or to terminate the process. The suspension time may be longer than requested due to the scheduling of other activity by the system.

RETURN VALUES

If the **nanosleep**() function returns because the requested time has elapsed, the value returned will be zero.

If the **nanosleep**() function returns due to the delivery of a signal, the value returned will be the −1, and the global variable *errno* will be set to indicate the interruption. If *rmtp* is non-NULL, the timespec structure it references is updated to contain the unslept amount (the request time minus the time actually slept).

ERRORS

If any of the following conditions occur, the **nanosleep** function shall return −1 and set *errno* to the corresponding value.

[EFAULT]	Either *rqtp* or *rmtp* points to memory that is not a valid part of the process address space.
[EINTR]	**nanosleep** was interrupted by the delivery of a signal.
[EINVAL]	*rqtp* specified a nanosecond value less than zero or greater than 1000 million.
[ENOSYS]	**nanosleep** is not supported by this implementation.

SEE ALSO

sleep(3)

STANDARDS

The **nanosleep**() function conforms to IEEE Std 1003.1b-1993 (“POSIX.1”).

NAME

nfssvc — NFS services

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>
#include <nfs/nfs.h>

int
nfssvc(*int flags*, *void *argstructp*);

DESCRIPTION

The **nfssvc**() function is used by the NFS daemons to pass information into and out of the kernel and also to enter the kernel as a server daemon. The *flags* argument consists of several bits that show what action is to be taken once in the kernel and the *argstructp* points to one of three structures depending on which bits are set in flags.

Calls used by nfsd(8)

On the server side, **nfssvc**() is called with the flag NFSSVC_NFSD and a pointer to a

```
struct nfsd_srvargs {
        struct nfsd    *nsd_nfsd;     /* Pointer to in kernel nfsd struct */
        uid_t          nsd_uid;       /* Effective uid mapped to cred */
        u_long         nsd_haddr;     /* Ip address of client */
        struct ucred   nsd_cr;        /* Cred. uid maps to */
        int            nsd_authlen;   /* Length of auth string (ret) */
        char           *nsd_authstr;  /* Auth string (ret) */
};
```

to enter the kernel as an nfsd(8) daemon. Whenever an nfsd(8) daemon receives a Kerberos authentication ticket, it will return from **nfssvc**() with errno set to ENEEDAUTH. The nfsd(8) will attempt to authenticate the ticket and generate a set of credentials on the server for the “user id” specified in the field nsd_uid. This is done by first authenticating the Kerberos ticket and then mapping the Kerberos principal to a local name and getting a set of credentials for that user via getpwnam(3) and getgrouplist(3). If successful, the nfsd(8) will call **nfssvc**() with the NFSSVC_NFSD and NFSSVC_AUTHIN flags set to pass the credential mapping in nsd_cr into the kernel to be cached on the server socket for that client. If the authentication failed, nfsd(8) calls **nfssvc**() with the flags NFSSVC_NFSD and NFSSVC_AUTHINFAIL to denote an authentication failure.

The master nfsd(8) server daemon calls **nfssvc**() with the flag NFSSVC_ADDSOCK and a pointer to a

```
struct nfsd_args {
        int     sock;          /* Socket to serve */
        caddr_t name;          /* Client address for connection based sockets */
        int     namelen;       /* Length of name */
};
```

to pass a server side NFS socket into the kernel for servicing by the nfsd(8) daemons.

Calls used by mountd(8)

The mountd(8) server daemon calls **nfssvc**() with the flag NFSSVC_SETEXPORTSLIST and a pointer to a *struct mountd_exports_list* object to atomically change the exports lists of a specific file system. This structure has the following fields:

*const char *mel_path*
: Path to the file system that will have its exports list replaced by the one described in the other fields.

size_t mel_nexports
: Number of valid entries in the *mel_export* field. If zero, the exports list will be cleared for the given file system.

struct export_args mel_export[AF_MAX]
: Set of exports to be used for the given file system.

RETURN VALUES

Usually **nfssvc** does not return unless the server is terminated by a signal when a value of 0 is returned. Otherwise, −1 is returned and the global variable *errno* is set to specify the error.

ERRORS

[ENEEDAUTH]	This special error value is really used for authentication support, particularly Kerberos, as explained above.
[EPERM]	The caller is not the super-user.

SEE ALSO

mount_nfs(8), nfsd(8)

HISTORY

The **nfssvc** function first appeared in 4.4BSD.

BUGS

The **nfssvc** system call is designed specifically for the NFS support daemons and as such is specific to their requirements. It should really return values to indicate the need for authentication support, since ENEEDAUTH is not really an error. Several fields of the argument structures are assumed to be valid and sometimes to be unchanged from a previous call, such that **nfssvc** must be used with extreme care.

NAME

ntp_adjtime, **ntp_gettime** — Network Time Protocol (NTP) daemon interface system calls

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/time.h>
#include <sys/timex.h>

int
ntp_adjtime(*struct timex* ∗);

int
ntp_gettime(*struct ntptimeval* ∗);

DESCRIPTION

The two system calls **ntp_adjtime**() and **ntp_gettime**() are the kernel interface to the Network Time Protocol (NTP) daemon ntpd(8).

The **ntp_adjtime**() function is used by the NTP daemon to adjust the system clock to an externally derived time. The time offset and related variables which are set by **ntp_adjtime**() are used by hardclock(9) to adjust the phase and frequency of the phase- or frequency-lock loop (PLL resp. FLL) which controls the system clock.

The **ntp_gettime**() function provides the time, maximum error (sync distance) and estimated error (dispersion) to client user application programs.

In the following, all variables that refer PPS are only relevant if the *PPS_SYNC* option (see options(4)) is enabled in the kernel.

ntp_adjtime() has as argument a *struct timex* ∗ of the following form:

```
struct timex {
        unsigned int modes;    /∗ clock mode bits (wo) ∗/
        long offset;           /∗ time offset (us) (rw) ∗/
        long freq;             /∗ frequency offset (scaled ppm) (rw) ∗/
        long maxerror;         /∗ maximum error (us) (rw) ∗/
        long esterror;         /∗ estimated error (us) (rw) ∗/
        int status;            /∗ clock status bits (rw) ∗/
        long constant;         /∗ pll time constant (rw) ∗/
        long precision;                /∗ clock precision (us) (ro) ∗/
        long tolerance;                /∗ clock frequency tolerance (scaled
                                ∗ ppm) (ro) ∗/
        /∗
         ∗ The following read-only structure members are implemented
         ∗ only if the PPS signal discipline is configured in the
         ∗ kernel.
         ∗/
        long ppsfreq;          /∗ pps frequency (scaled ppm) (ro) ∗/
        long jitter;           /∗ pps jitter (us) (ro) ∗/
        int shift;             /∗ interval duration (s) (shift) (ro) ∗/
        long stabil;           /∗ pps stability (scaled ppm) (ro) ∗/
        long jitcnt;           /∗ jitter limit exceeded (ro) ∗/
        long calcnt;           /∗ calibration intervals (ro) ∗/
        long errcnt;           /* calibration errors (ro) */
        long stbcnt;           /* stability limit exceeded (ro) */
};
```

The members of this struct have the following meanings when used as argument for **ntp_adjtime**():

modes Defines what settings should be changed with the current **ntp_adjtime**() call (write-only). Bitwise OR of the following:

MOD_OFFSET	set time offset
MOD_FREQUENCY	set frequency offset
MOD_MAXERROR	set maximum time error
MOD_ESTERROR	set estimated time error
MOD_STATUS	set clock status bits
MOD_TIMECONST	set PLL time constant
MOD_CLKA	set clock A
MOD_CLKB	set clock B

offset Time offset (in microseconds), used by the PLL/FLL to adjust the system time in small increments (read-write).

freq Frequency offset (scaled ppm) (read-write).

maxerror Maximum error (in microseconds). Initialized by an **ntp_adjtime**() call, and increased by the kernel once each second to reflect the maximum error bound growth (read-write).

esterror Estimated error (in microseconds). Set and read by **ntp_adjtime**(), but unused by the kernel (read-write).

status System clock status bits (read-write). Bitwise OR of the following:

STA_PLL	Enable PLL updates (read-write).
STA_PPSFREQ	Enable PPS freq discipline (read-write).
STA_PPSTIME	Enable PPS time discipline (read-write).
STA_FLL	Select frequency-lock mode (read-write).
STA_INS	Insert leap (read-write).
STA_DEL	Delete leap (read-write).
STA_UNSYNC	Clock unsynchronized (read-write).
STA_FREQHOLD	Hold frequency (read-write).
STA_PPSSIGNAL	PPS signal present (read-only).
STA_PPSJITTER	PPS signal jitter exceeded (read-only).
STA_PPSWANDER	PPS signal wander exceeded (read-only).
STA_PPSERROR	PPS signal calibration error (read-only).
STA_CLOCKERR	Clock hardware fault (read-only).

constant PLL time constant, determines the bandwidth, or “stiffness”, of the PLL (read-write).

precision Clock precision (in microseconds). In most cases the same as the kernel tick variable (see hz(9)). If a precision clock counter or external time-keeping signal is available, it could be much lower (and depend on the state of the signal) (read-only).

tolerance Maximum frequency error, or tolerance of the CPU clock oscillator (scaled ppm). Ordinarily a property of the architecture, but could change under the influence of external time-keeping signals (read-only).

ppsfreq PPS frequency offset produced by the frequency median filter (scaled ppm) (read-only).

jitter PPS jitter measured by the time median filter in microseconds (read-only).

shift	Logarithm to base 2 of the interval duration in seconds (PPS, read-only).
stabil	PPS stability (scaled ppm); dispersion (wander) measured by the frequency median filter (read-only).
jitcnt	Number of seconds that have been discarded because the jitter measured by the time median filter exceeded the limit *MAXTIME* (PPS, read-only).
calcnt	Count of calibration intervals (PPS, read-only).
errcnt	Number of calibration intervals that have been discarded because the wander exceeded the limit *MAXFREQ* or where the calibration interval jitter exceeded two ticks (PPS, read-only).
stbcnt	Number of calibration intervals that have been discarded because the frequency wander exceeded the limit *MAXFREQ*/4 (PPS, read-only).

After the **ntp_adjtime**() call, the *struct timex* ∗ structure contains the current values of the corresponding variables.

ntp_gettime() has as argument a *struct ntptimeval* ∗ with the following members:

```
struct ntptimeval {
        struct timespec time;  /* current time (ro) */
        long maxerror;         /* maximum error (us) (ro) */
        long esterror;         /* estimated error (us) (ro) */
        /* the following are placeholders for now */
        long tai;              /* TAI offset */
        int time_state;                /* time status */
};
```

These have the following meaning:

time	Current time (read-only).
maxerror	Maximum error in microseconds (read-only).
esterror	Estimated error in microseconds (read-only).

RETURN VALUES

ntp_adjtime() and **ntp_gettime**() return the current state of the clock on success, or any of the errors of copyin(9) and copyout(9). **ntp_adjtime**() may additionally return EPERM if the user calling **ntp_adjtime**() does not have sufficient permissions.

Possible states of the clock are:

TIME_OK	Everything okay, no leap second warning.
TIME_INS	“insert leap second” warning.
TIME_DEL	“delete leap second” warning.
TIME_OOP	Leap second in progress.
TIME_WAIT	Leap second has occurred.
TIME_ERROR	Clock not synchronized.

SEE ALSO

options(4), ntpd(8), hardclock(9), hz(9)

J. Mogul, D. Mills, J. Brittenson, J. Stone, and U. Windl, *Pulse-Per-Second API for UNIX-like Operating Systems*, RFC 2783, March 2000.

NAME

open — open or create a file for reading or writing

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <fcntl.h>

int
open(*const char *path*, *int flags*, *mode_t mode*);

DESCRIPTION

The file name specified by *path* is opened for reading and/or writing as specified by the argument *flags* and the file descriptor returned to the calling process. The *flags* are specified by *or*'ing the values listed below. Applications must specify exactly one of the first three values (file access methods):

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.

Any combination of the following may be used:

O_NONBLOCK	Do not block on open or for data to become available.
O_APPEND	Append to the file on each write.
O_CREAT	Create the file if it does not exist, in which case the file is created with mode *mode* as described in chmod(2) and modified by the process' umask value (see umask(2)).
O_TRUNC	Truncate size to 0.
O_EXCL	Error if O_CREAT and the file already exists.
O_SHLOCK	Atomically obtain a shared lock.
O_EXLOCK	Atomically obtain an exclusive lock.
O_NOFOLLOW	If last path element is a symlink, don't follow it. This option is provided for compatibility with other operating systems, but its security value is questionable.
O_DSYNC	If set, write operations will be performed according to synchronized I/O data integrity completion: each write will wait for the file data to be committed to stable storage.
O_SYNC	If set, write operations will be performed according to synchronized I/O file integrity completion: each write will wait for both the file data and file status to be committed to stable storage.
O_RSYNC	If set, read operations will complete at the same level of integrity which is in effect for write operations: if specified together with O_SYNC, each read will wait for the file status to be committed to stable storage. Combining O_RSYNC with O_DSYNC only, or specifying it without any other synchronized I/O integrity completion flag set, has no further effect.
O_ALT_IO	Alternate I/O semantics will be used for read and write operations on the file descriptor. Alternate semantics are defined by the underlying layers and will not have any alternate effect in most cases.

O_NOCTTY	If the file is a terminal device, the opened device is not made the controlling terminal for the session. This flag has no effect on NetBSD, since the system defaults to the abovementioned behaviour. The flag is present only for standards conformance.
O_DIRECT	If set on a regular file, data I/O operations will not buffer the data being transferred in the kernel's cache, but rather transfer the data directly between user memory and the underlying device driver if possible. This flag is advisory; the request may be performed in the normal buffered fashion if certain conditions are not met, e.g. if the request is not sufficiently aligned or if the file is mapped. To meet the alignment requirements for direct I/O, the file offset, the length of the I/O and the address of the buffer in memory must all be multiples of DEV_BSIZE (512 bytes). If the I/O request is made using an interface that supports scatter/gather via struct iovec, each element of the request must meet the above alignment constraints.

Opening a file with O_APPEND set causes each write on the file to be appended to the end. If O_TRUNC is specified and the file exists, the file is truncated to zero length.

If O_EXCL is set with O_CREAT and the file already exists, **open**() returns an error. This may be used to implement a simple exclusive access locking mechanism. If O_EXCL is set and the last component of the pathname is a symbolic link, **open**() will fail even if the symbolic link points to a non-existent name.

If the O_NONBLOCK flag is specified, do not wait for the device or file to be ready or available. If the **open**() call would result in the process being blocked for some reason (e.g., waiting for carrier on a dialup line), **open**() returns immediately. This flag also has the effect of making all subsequent I/O on the open file non-blocking.

When opening a file, a lock with flock(2) semantics can be obtained by setting O_SHLOCK for a shared lock, or O_EXLOCK for an exclusive lock. If creating a file with O_CREAT, the request for the lock will never fail (provided that the underlying filesystem supports locking).

If **open**() is successful, the file pointer used to mark the current position within the file is set to the beginning of the file.

When a new file is created it is given the group of the directory which contains it.

The new descriptor is set to remain open across execve(2) system calls; see close(2) and fcntl(2).

The system imposes a limit on the number of file descriptors open simultaneously by one process. Calling getdtablesize(3) returns the current system limit.

RETURN VALUES

If successful, **open**() returns a non-negative integer, termed a file descriptor. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

The named file is opened unless:

[EPERM]	The file's flags (see chflags(2)) don't allow the file to be opened.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded NAME_MAX characters, or an entire path name exceeded PATH_MAX characters.
[ENOENT]	O_CREAT is not set and the named file does not exist, or a component of the path name that must exist does not exist.

[EACCES]	Search permission is denied for a component of the path prefix, the required permissions (for reading and/or writing) are denied for the given flags, or `O_CREAT` is specified, the file does not exist, and the directory in which it is to be created does not permit writing.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EISDIR]	The named file is a directory, and the arguments specify it is to be opened for writing.
[EROFS]	The named file resides on a read-only file system, and the file is to be modified.
[EMFILE]	The process has already reached its limit for open file descriptors.
[ENFILE]	The system file table is full.
[ENXIO]	The named file is a character special or block special file, and the device associated with this special file does not exist, or the named file is a FIFO, `O_NONBLOCK` and `O_WRONLY` is set and no process has the file open for reading.
[EINTR]	The **open**() operation was interrupted by a signal.
[EOPNOTSUPP]	`O_SHLOCK` or `O_EXLOCK` is specified but the underlying filesystem does not support locking.
[ENOSPC]	`O_CREAT` is specified, the file does not exist, and the directory in which the entry for the new file is being placed cannot be extended because there is no space left on the file system containing the directory.
[ENOSPC]	`O_CREAT` is specified, the file does not exist, and there are no free inodes on the file system on which the file is being created.
[EDQUOT]	`O_CREAT` is specified, the file does not exist, and the directory in which the entry for the new file is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
[EDQUOT]	`O_CREAT` is specified, the file does not exist, and the user's quota of inodes on the file system on which the file is being created has been exhausted.
[EIO]	An I/O error occurred while making the directory entry or allocating the inode for `O_CREAT`.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed and the **open**() call requests write access.
[EFAULT]	*path* points outside the process's allocated address space.
[EEXIST]	`O_CREAT` and `O_EXCL` were specified and the file exists.
[EOPNOTSUPP]	An attempt was made to open a socket (not currently implemented).

SEE ALSO

chmod(2), close(2), dup(2), lseek(2), read(2), umask(2), write(2), getdtablesize(3)

STANDARDS

The **open**() function conforms to ISO/IEC 9945-1:1990 ("POSIX.1"). The *flags* values `O_DSYNC`, `O_SYNC` and `O_RSYNC` are extensions defined in IEEE Std 1003.1b-1993 ("POSIX.1").

The `O_SHLOCK`, `O_EXLOCK`, and `O_NOFOLLOW` flags are non-standard extensions and should not be used if portability is of concern.

HISTORY

An **open**() function call appeared in Version 6 AT&T UNIX.

NAME

pathconf, **fpathconf** — get configurable pathname variables

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

long
pathconf(*const char *path*, *int name*);

long
fpathconf(*int fd*, *int name*);

DESCRIPTION

The **pathconf**() and **fpathconf**() functions provide a method for applications to determine the current value of a configurable system limit or option variable associated with a pathname or file descriptor.

For **pathconf**, the *path* argument is the name of a file or directory. For **fpathconf**, the *fd* argument is an open file descriptor. The *name* argument specifies the system variable to be queried. Symbolic constants for each name value are found in the include file `<unistd.h>`.

The available values are as follows:

`_PC_LINK_MAX`
: The maximum file link count.

`_PC_MAX_CANON`
: The maximum number of bytes in terminal canonical input line.

`_PC_MAX_INPUT`
: The minimum maximum number of bytes for which space is available in a terminal input queue.

`_PC_NAME_MAX`
: The maximum number of bytes in a filename, not including a terminating null character.

`_PC_PATH_MAX`
: The maximum number of bytes in a pathname, including the terminating null character.

`_PC_PIPE_BUF`
: The maximum number of bytes which will be written atomically to a pipe.

`_PC_CHOWN_RESTRICTED`
: Return 1 if appropriate privileges are required for the `chown`(2) system call, otherwise 0.

`_PC_NO_TRUNC`
: Return 1 if filenames longer than `{NAME_MAX}` are truncated.

`_PC_VDISABLE`
: Returns the terminal character disabling value.

`_PC_SYNC_IO`
: Returns 1 if synchronized I/O is supported, otherwise 0.

`_PC_FILESIZEBITS`
: If the maximum size file that could ever exist on the mounted file system is `maxsize`, then the returned value is 2 plus the floor of the base 2 logarithm of `maxsize`.

RETURN VALUES

If the call to **pathconf** or **fpathconf** is not successful, −1 is returned and *errno* is set appropriately. Otherwise, if the variable is associated with functionality that does not have a limit in the system, −1 is returned and *errno* is not modified. Otherwise, the current variable value is returned.

ERRORS

If any of the following conditions occur, the **pathconf** and **fpathconf** functions shall return −1 and set *errno* to the corresponding value.

[EINVAL]	The value of the *name* argument is invalid.
[EINVAL]	The implementation does not support an association of the variable name with the associated file.

pathconf() will fail if:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EIO]	An I/O error occurred while reading from or writing to the file system.

fpathconf() will fail if:

[EBADF]	*fd* is not a valid open file descriptor.
[EIO]	An I/O error occurred while reading from or writing to the file system.

SEE ALSO

sysctl(3)

STANDARDS

The **pathconf**() and **fpathconf**() functions conform to ISO/IEC 9945-1:1990 (“POSIX.1”).

HISTORY

The **pathconf** and **fpathconf** functions first appeared in 4.4BSD.

NAME

pipe — create descriptor pair for interprocess communication

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
pipe(*int fildes[2]*);

DESCRIPTION

The **pipe**() function creates a *pipe*, which is an object allowing unidirectional data flow, and allocates a pair of file descriptors. The first descriptor connects to the *read end* of the pipe, and the second connects to the *write end*, so that data written to *fildes[1]* appears on (i.e., can be read from) *fildes[0]*. This allows the output of one program to be sent to another program: the source's standard output is set up to be the write end of the pipe, and the sink's standard input is set up to be the read end of the pipe. The pipe itself persists until all its associated descriptors are closed.

A pipe whose read or write end has been closed is considered *widowed*. Writing on such a pipe causes the writing process to receive a SIGPIPE signal. Widowing a pipe is the only way to deliver end-of-file to a reader: after the reader consumes any buffered data, reading a widowed pipe returns a zero count.

RETURN VALUES

On successful creation of the pipe, zero is returned. Otherwise, a value of −1 is returned and the variable *errno* set to indicate the error.

ERRORS

The **pipe**() call will fail if:

[EMFILE]	Too many descriptors are active.
[ENFILE]	The system file table is full.
[EFAULT]	The *fildes* buffer is in an invalid area of the process's address space. The reliable detection of this error cannot be guaranteed; when not detected, a signal may be delivered to the process, indicating an address violation.

SEE ALSO

sh(1), fork(2), read(2), socketpair(2), write(2)

STANDARDS

The **pipe**() function conforms to ISO/IEC 9945-1:1990 ("POSIX.1").

HISTORY

A **pipe**() function call appeared in Version 6 AT&T UNIX.

NAME

pmc_control, **pmc_get_info** — Hardware Performance Monitoring Interface

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/pmc.h>

int
pmc_control(*int ctr*, *int op*, *void ∗argp*);

int
pmc_get_info(*int ctr*, *int op*, *void ∗argp*);

DESCRIPTION

pmc_get_info() returns the number of counters in the system or information on a specified counter *ctr*. The possible values for *op* are:

PMC_INFO_NCOUNTERS
: When querying the number of counters in the system, *ctr* is ignored and *argp* is of type *int ∗*. Upon return, the integer pointed to by *argp* will contain the number of counters that are available in the system.

PMC_INFO_CPUCTR_TYPE
: When querying the type of a counter in the system, *ctr* refers to the counter being queried, and *argp* is of type *int ∗*. Upon return, the integer pointed to by *argp* will contain the implementation-dependent type of the specified counter.

  If *ctr* is −1, the integer pointed to by *argp* will contain the machine-dependent type describing the CPU or counter configuration.

PMC_INFO_COUNTER_VALUE
: When querying the value of a counter in the system, *ctr* refers to the counter being queried, and *argp* is of type *uint64_t ∗*. Upon return, the 64-bit integer pointed to by *argp* will contain the value of the specified counter.

PMC_INFO_ACCUMULATED_COUNTER_VALUE
: When querying the value of a counter in the system, *ctr* refers to the counter being queried, and *argp* is of type *uint64_t ∗*. Upon return, the 64-bit integer pointed to by *argp* will contain the sum of the accumulated values of specified counter in all exited subprocesses of the current process.

pmc_control() manipulates the specified counter *ctr* in one of several fashions. The *op* parameter determines the action taken by the kernel and also the interpretation of the *argp* parameter. The possible values for *op* are:

PMC_OP_START
: Starts the specified *ctr* running. It must be preceded by a call with PMC_OP_CONFIGURE. *argp* is ignored in this case and may be NULL.

PMC_OP_STOP
: Stops the specified *ctr* from running. *argp* is ignored in this case and may be NULL.

PMC_OP_CONFIGURE
: Configures the specified *ctr* prior to running. *argp* is a pointer to a *struct pmc_counter_cfg*.

```
struct pmc_counter_cfg {
        pmc_evid_t     event_id;
        pmc_ctr_t      reset_value;
        uint32_t       flags;
};
```

`event_id`
is the event ID to be counted.

`reset_value`
is a value to which the counter should be reset on overflow (if supported by the implementation). This is most useful when profiling (see `PMC_OP_PROFSTART`, below). This value is defined to be the number of counter ticks before the next overflow. So, to get a profiling tick on every hundredth data cache miss, set the `event_id` to the proper value for "dcache-miss" and set `reset_value` to 100.

`flags` Currently unused.

`PMC_OP_PROFSTART`
Configures the specified *`ctr`* for use in profiling. *`argp`* is a pointer to a *`struct pmc_counter_cfg`* as in `PMC_OP_CONFIGURE`, above. This request allocates a kernel counter, which will fail if any process is using the requested counter. Not all implementations or counters may support this option.

`PMC_OP_PROFSTOP`
Stops the specified *`ctr`* from being used for profiling. *`argp`* is ignored in this case and may be `NULL`.

RETURN VALUES

A return value of 0 indicates that the call succeeded. Otherwise, −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

Among the possible error codes from **`pmc_control`**() and **`pmc_get_info`**() are

[EFAULT]	The address specified for the *`argp`* is invalid.
[ENXIO]	Specified counter is not yet configured.
[EINPROGRESS]	`PMC_OP_START` was passed for a counter that is already running.
[EINVAL]	Specified counter was invalid.
[EBUSY]	If the requested counter is already in use--either by the current process or by the kernel.
[ENODEV]	If and only if the specified counter event is not valid for the specified counter when configuring a counter or starting profiling.
[ENOMEM]	If the kernel is unable to allocate memory.

SEE ALSO

pmc(1), pmc(9)

HISTORY

The **`pmc_control`**() and **`pmc_get_info`**() system calls appeared in NetBSD 2.0.

NAME

poll, pollts — synchronous I/O multiplexing

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <poll.h>

int
poll(*struct pollfd ∗fds*, *nfds_t nfds*, *int timeout*);

#include <poll.h>
#include <signal.h>
#include <time.h>

int
pollts(*struct pollfd ∗ restrict fds*, *nfds_t nfds*,
    const struct timespec ∗ restrict ts,
    const sigset_t ∗ restrict sigmask);

DESCRIPTION

poll() and **pollts**() examine a set of file descriptors to see if some of them are ready for I/O. The *fds* argument is a pointer to an array of pollfd structures as defined in ⟨poll.h⟩ (shown below). The *nfds* argument determines the size of the *fds* array.

```
struct pollfd {
    int    fd;       /∗ file descriptor ∗/
    short  events;   /∗ events to look for ∗/
    short  revents;  /∗ events returned ∗/
};
```

The fields of *struct pollfd* are as follows:

fd	File descriptor to poll. If the value in *fd* is negative, the file descriptor is ignored and *revents* is set to 0.
events	Events to poll for. (See below.)
revents	Events which may occur. (See below.)

The event bitmasks in *events* and *revents* have the following bits:

POLLIN	Data other than high priority data may be read without blocking.
POLLRDNORM	Normal data may be read without blocking.
POLLRDBAND	Data with a non-zero priority may be read without blocking.
POLLPRI	High priority data may be read without blocking.
POLLOUT	Normal data may be written without blocking.
POLLWRNORM	Equivalent to POLLOUT.
POLLWRBAND	Data with a non-zero priority may be written without blocking.
POLLERR	An exceptional condition has occurred on the device or socket. This flag is always checked, even if not present in the *events* bitmask.

POLLHUP	The device or socket has been disconnected. This flag is always checked, even if not present in the *events* bitmask. Note that POLLHUP and POLLOUT should never be present in the *revents* bitmask at the same time. If the remote end of a socket is closed, **poll**() returns a POLLIN event, rather than a POLLHUP.
POLLNVAL	The file descriptor is not open. This flag is always checked, even if not present in the *events* bitmask.

If *timeout* is neither zero nor INFTIM (−1), it specifies a maximum interval to wait for any file descriptor to become ready, in milliseconds. If *timeout* is INFTIM (−1), the poll blocks indefinitely. If *timeout* is zero, then **poll**() will return without blocking.

If *ts* is a non-null pointer, it references a timespec structure which specifies a maximum interval to wait for any file descriptor to become ready. If *ts* is a null pointer, **pollts**() blocks indefinitely. If *ts* is a non-null pointer, referencing a zero-valued timespec structure, then **pollts**() will return without blocking.

If *sigmask* is a non-null pointer, then the **pollts**() function shall replace the signal mask of the caller by the set of signals pointed to by *sigmask* before examining the descriptors, and shall restore the signal mask of the caller before returning.

RETURN VALUES

poll() returns the number of descriptors that are ready for I/O, or −1 if an error occurred. If the time limit expires, **poll**() returns 0. If **poll**() returns with an error, including one due to an interrupted call, the *fds* array will be unmodified.

COMPATIBILITY

This implementation differs from the historical one in that a given file descriptor may not cause **poll**() to return with an error. In cases where this would have happened in the historical implementation (e.g. trying to poll a revoke(2)d descriptor), this implementation instead copies the *events* bitmask to the *revents* bitmask. Attempting to perform I/O on this descriptor will then return an error. This behaviour is believed to be more useful.

ERRORS

An error return from **poll**() indicates:

[EFAULT]	*fds* points outside the process's allocated address space.
[EINTR]	A signal was delivered before the time limit expired and before any of the selected events occurred.
[EINVAL]	The specified time limit is negative.

SEE ALSO

accept(2), connect(2), read(2), recv(2), select(2), send(2), write(2)

HISTORY

The **poll**() function appeared in AT&T System V.3 UNIX. The **pollts**() function first appeared in NetBSD 3.0.

BUGS

The distinction between some of the fields in the *events* and *revents* bitmasks is really not useful without STREAMS. The fields are defined for compatibility with existing software.

NAME

posix_fadvise — hint at the future access pattern of a file

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/fcntl.h>

int
posix_fadvise(*int fd*, *off_t offset*, *off_t size*, *int hint*);

DESCRIPTION

posix_fadvise() hints at the application's access pattern to the file and range given by the file descriptor, *fd*, and *offset* and *size*. If *size* is zero, it means to the end of file.

hint should be one of the followings.

POSIX_FADV_NORMAL	No hint. (default)
POSIX_FADV_RANDOM	Random access.
POSIX_FADV_SEQUENTIAL	Sequential access. (from lower offset to higher offset.)
POSIX_FADV_WILLNEED	Will be accessed.
POSIX_FADV_DONTNEED	Will not be accessed.
POSIX_FADV_NOREUSE	Will be accessed just once.

Calling **posix_fadvise**() doesn't alter the semantics of the operations, it is only a matter of performance.

RETURN VALUES

On success, **posix_fadvise**() returns 0. Otherwise, it returns an error number.

SEE ALSO

errno(2), madvise(2)

BUGS

POSIX_FADV_WILLNEED, POSIX_FADV_DONTNEED, and POSIX_FADV_NOREUSE are not implemented.

For POSIX_FADV_NORMAL, POSIX_FADV_RANDOM, and POSIX_FADV_SEQUENTIAL, the current implementation ignores *offset* and *size*, and applies the hint to the whole file.

NAME

profil — control process profiling

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
profil(*char *samples*, *size_t size*, *u_long offset*, *u_int scale*);

DESCRIPTION

The **profil**() function enables or disables program counter profiling of the current process. If profiling is enabled, then at every clock tick, the kernel updates an appropriate count in the *samples* buffer.

The buffer *samples* contains *size* bytes and is divided into a series of 16-bit bins. Each bin counts the number of times the program counter was in a particular address range in the process when a clock tick occurred while profiling was enabled. For a given program counter address, the number of the corresponding bin is given by the relation:

```
[(pc - offset) / 2] * scale / 65536
```

The *offset* parameter is the lowest address at which the kernel takes program counter samples. The *scale* parameter ranges from 1 to 65536 and can be used to change the span of the bins. A scale of 65536 maps each bin to 2 bytes of address range; a scale of 32768 gives 4 bytes, 16384 gives 8 bytes and so on. Intermediate values provide approximate intermediate ranges. A *scale* value of 0 disables profiling.

RETURN VALUES

If the *scale* value is nonzero and the buffer *samples* contains an illegal address, **profil**() returns −1, profiling is terminated and *errno* is set appropriately. Otherwise **profil**() returns 0.

FILES

/usr/lib/gcrt0.o	profiling C run-time startup file
gmon.out	conventional name for profiling output file. This may be different if the PROFDIR environment variable is set.

ERRORS

The following error may be reported:

[EFAULT]	The buffer *samples* contains an invalid address.

SEE ALSO

gprof(1), moncontrol(3)

BUGS

This routine should be named **profile**().

The *samples* argument should really be a vector of type *unsigned short*.

The format of the gmon.out file is undocumented.

NAME

ptrace — process tracing and debugging

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/ptrace.h>

int
ptrace(*int request*, *pid_t pid*, *void *addr*, *int data*);

DESCRIPTION

ptrace() provides tracing and debugging facilities. It allows one process (the *tracing* process) to control another (the *traced* process). Most of the time, the traced process runs normally, but when it receives a signal (see sigaction(2)), it stops. The tracing process is expected to notice this via wait(2) or the delivery of a SIGCHLD signal, examine the state of the stopped process, and cause it to terminate or continue as appropriate. **ptrace**() is the mechanism by which all this happens.

The *request* argument specifies what operation is being performed; the meaning of the rest of the arguments depends on the operation, but except for one special case noted below, all **ptrace**() calls are made by the tracing process, and the *pid* argument specifies the process ID of the traced process. *request* can be:

PT_TRACE_ME
: This request is the only one used by the traced process; it declares that the process expects to be traced by its parent. All the other arguments are ignored. (If the parent process does not expect to trace the child, it will probably be rather confused by the results; once the traced process stops, it cannot be made to continue except via **ptrace**().) When a process has used this request and calls execve(2) or any of the routines built on it (such as execv(3)), it will stop before executing the first instruction of the new image. Also, any setuid or setgid bits on the executable being executed will be ignored.

PT_READ_I, PT_READ_D
: These requests read a single int of data from the traced process' address space. Traditionally, **ptrace**() has allowed for machines with distinct address spaces for instruction and data, which is why there are two requests: conceptually, PT_READ_I reads from the instruction space and PT_READ_D reads from the data space. In the current NetBSD implementation, these two requests are completely identical. The *addr* argument specifies the address (in the traced process' virtual address space) at which the read is to be done. This address does not have to meet any alignment constraints. The value read is returned as the return value from **ptrace**().

PT_WRITE_I, PT_WRITE_D
: These requests parallel PT_READ_I and PT_READ_D, except that they write rather than read. The *data* argument supplies the value to be written.

PT_CONTINUE
: The traced process continues execution. *addr* is an address specifying the place where execution is to be resumed (a new value for the program counter), or (caddr_t)1 to indicate that execution is to pick up where it left off. *data* provides a signal number to be delivered to the traced process as it resumes execution, or 0 if no signal is to be sent.

PT_KILL The traced process terminates, as if PT_CONTINUE had been used with SIGKILL given as the signal to be delivered.

PT_ATTACH This request allows a process to gain control of an otherwise unrelated process and begin tracing it. It does not need any cooperation from the to-be-traced process. In this case, *pid* specifies the process ID of the to-be-traced process, and the other two arguments are ignored. This request requires that the target process must have the same real UID as the tracing process, and that it must not be executing a setuid or setgid executable. (If the tracing process is running as root, these restrictions do not apply.) The tracing process will see the newly-traced process stop and may then control it as if it had been traced all along.

Three other restrictions apply to all tracing processes, even those running as root. First, no process may trace a system process. Second, no process may trace the process running init(8). Third, if a process has its root directory set with chroot(2), it may not trace another process unless that process's root directory is at or below the tracing process's root.

PT_DETACH This request is like PT_CONTINUE, except that after it succeeds, the traced process is no longer traced and continues execution normally.

PT_IO This request is a more general interface that can be used instead of PT_READ_D, PT_WRITE_D, PT_READ_I, and PT_WRITE_I. The I/O request is encoded in a "struct ptrace_io_desc" defined as:

```
struct ptrace_io_desc {
        int     piod_op;
        void    *piod_offs;
        void    *piod_addr;
        size_t  piod_len;
};
```

where *piod_offs* is the offset within the traced process where the I/O operation should take place, *piod_addr* is the buffer in the tracing process, and *piod_len* is the length of the I/O request. The *piod_op* field specifies which type of I/O operation to perform. Possible values are:

PIOD_READ_D

PIOD_WRITE_D

PIOD_READ_I

PIOD_WRITE_I

See the description of PT_READ_I for the difference between I and D spaces. A pointer to the I/O descriptor is passed in the *addr* argument to **ptrace**(). On return, the *piod_len* field in the I/O descriptor will be updated with the actual number of bytes transferred. If the requested I/O could not be successfully performed, **ptrace**() will return −1 and set *errno*.

PT_DUMPCORE
Makes the process specified in the *pid* pid generate a core dump. The *addr* argument should contain the name of the core file to be generated and the *data* argument should contain the length of the core filename. This **ptrace** call currently does not stop the child process so it can generate inconsistent data.

PT_LWPINFO Returns information about the specific thread from the process specified in the *pid* argument. The *addr* argument should contain a "struct ptrace_lwpinfo" defined as:

```
struct ptrace_lwpinfo {
        lwpid_t pl_lwpid;
        int pl_event;
};
```

where *pl_lwpid* contains the thread for which to get info. Upon return *pl_event* contains the event that stopped the thread. Possible values are:

`PL_EVENT_NONE`

`PL_EVENT_SIGNAL`

The *data* argument should contain “`sizeof(struct ptrace_lwpinfo)`”.

`PT_SYSCALL` Stops a process before and after executing each system call.

Additionally, the following requests exist but are not available on all machine architectures. The file ⟨`machine/ptrace.h`⟩ lists which requests exist on a given machine.

`PT_STEP` Execution continues as in request PT_CONTINUE; however as soon as possible after execution of at least one instruction, execution stops again.

`PT_GETREGS` This request reads the traced process’ machine registers into the “`struct reg`” (defined in ⟨`machine/reg.h`⟩) pointed to by *addr*.

`PT_SETREGS` This request is the converse of `PT_GETREGS`; it loads the traced process’ machine registers from the “`struct reg`” (defined in ⟨`machine/reg.h`⟩) pointed to by *addr*.

`PT_GETFPREGS`
This request reads the traced process’ floating-point registers into the “`struct fpreg`” (defined in ⟨`machine/reg.h`⟩) pointed to by *addr*.

`PT_SETFPREGS`
This request is the converse of `PT_GETFPREGS`; it loads the traced process’ floating-point registers from the “`struct fpreg`” (defined in ⟨`machine/reg.h`⟩) pointed to by *addr*.

`PT_DUMPCORE`
Cause the traced process to dump core. If the *addr* argument is not `NULL` it is taken to be the pathname of the core file to be generated and the *data* argument should contain the length of the pathname. The pathname may contain `%` patterns that are expanded as described in `sysctl`(8). If the *data* argument is `NULL`, the default core file path generation rules are followed.

ERRORS

Some requests can cause **ptrace**() to return `−1` as a non-error value; to disambiguate, *errno* can be set to 0 before the call and checked afterwards. The possible errors are:

`[EAGAIN]`
Process is currently exec’ing and cannot be traced.

`[ESRCH]`
No process having the specified process ID exists.

`[EINVAL]`
- A process attempted to use `PT_ATTACH` on itself.
- The *request* was not a legal request on this machine architecture.

- The signal number (in *data*) to `PT_CONTINUE` was neither 0 nor a legal signal number.
- `PT_GETREGS`, `PT_SETREGS`, `PT_GETFPREGS`, or `PT_SETFPREGS` was attempted on a process with no valid register set. (This is normally true only of system processes.)

[EBUSY]
- `PT_ATTACH` was attempted on a process that was already being traced.
- A request attempted to manipulate a process that was being traced by some process other than the one making the request.
- A request (other than `PT_ATTACH`) specified a process that wasn't stopped.

[EPERM]
- A request (other than `PT_ATTACH`) attempted to manipulate a process that wasn't being traced at all.
- An attempt was made to use `PT_ATTACH` on a process in violation of the requirements listed under `PT_ATTACH` above.

SEE ALSO

`sigaction`(2), `signal`(7)

BUGS

On the SPARC, the PC is set to the provided PC value for `PT_CONTINUE` and similar calls, but the NPC is set willy-nilly to 4 greater than the PC value. Using `PT_GETREGS` and `PT_SETREGS` to modify the PC, passing `(caddr_t)1` to **ptrace**(), should be able to sidestep this.

NAME

quotactl — manipulate filesystem quotas

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <ufs/ufs/quota.h>

int
quotactl(*const char *path*, *int cmd*, *int id*, *void *addr*);

DESCRIPTION

The **quotactl**() call enables, disables and manipulates filesystem quotas. A quota control command given by *cmd* operates on the given filename *path* for the given user *id*. The address of an optional command specific data structure, *addr*, may be given; its interpretation is discussed below with each command.

Currently quotas are supported only for the ‘‘ffs’’ and ‘‘lfs’’ filesystem. For both of them, a command is composed of a primary command (see below) and a command type used to interpret the *id*. Types are supported for interpretation of user identifiers and group identifiers. The ‘‘ffs’’ and ‘‘lfs’’ specific commands are:

`Q_QUOTAON` Enable disk quotas for the filesystem specified by *path*. The command type specifies the type of the quotas being enabled. The *addr* argument specifies a file from which to take the quotas. The quota file must exist; it is normally created with the `quotacheck`(8) program. The *id* argument is unused. Only the super-user may turn quotas on.

`Q_QUOTAOFF`
Disable disk quotas for the filesystem specified by *path*. The command type specifies the type of the quotas being disabled. The *addr* and *id* arguments are unused. Only the super-user may turn quotas off.

`Q_GETQUOTA`
Get disk quota limits and current usage for the user or group (as determined by the command type) with identifier *id*. *addr* is a pointer to a *struct dqblk* structure (defined in 〈`ufs/ufs/quota.h`〉).

`Q_SETQUOTA`
Set disk quota limits for the user or group (as determined by the command type) with identifier *id*. *addr* is a pointer to a *struct dqblk* structure (defined in 〈`ufs/ufs/quota.h`〉). The usage fields of the *dqblk* structure are ignored. This call is restricted to the super-user.

`Q_SETUSE` Set disk usage limits for the user or group (as determined by the command type) with identifier *id*. *addr* is a pointer to a *struct dqblk* structure (defined in 〈`ufs/ufs/quota.h`〉). Only the usage fields are used. This call is restricted to the super-user.

`Q_SYNC` Update the on-disk copy of quota usages. The command type specifies which type of quotas are to be updated. The *id* and *addr* parameters are ignored.

RETURN VALUES

A successful call returns 0, otherwise the value −1 is returned and the global variable *errno* indicates the reason for the failure.

ERRORS

A **quotactl**() call will fail if:

[EOPNOTSUPP]	The kernel has not been compiled with the QUOTA option.
[EUSERS]	The quota table cannot be expanded.
[EINVAL]	*cmd* or the command type is invalid.
[EACCES]	In Q_QUOTAON, the quota file is not a plain file, or search permission is denied for a component of a path prefix.
[ENOTDIR]	A component of a path prefix was not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	A filename does not exist.
[ELOOP]	Too many symbolic links were encountered in translating a pathname.
[EROFS]	In Q_QUOTAON, the quota file resides on a read-only filesystem.
[EIO]	An I/O error occurred while reading from or writing to a file containing quotas.
[EFAULT]	*path* points outside the process's allocated address space, or an invalid *addr* was supplied; the associated structure could not be copied in or out of the kernel.
[EPERM]	The call was privileged and the caller was not the super-user.

SEE ALSO

quota(1), fstab(5), edquota(8), quotacheck(8), quotaon(8), repquota(8)

HISTORY

The **quotactl**() function call appeared in 4.3BSD−Reno.

BUGS

There should be some way to integrate this call with the resource limit interface provided by setrlimit(2) and getrlimit(2).

NAME

rasctl — restartable atomic sequences

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/ras.h>

int
rasctl(*void *addr*, *size_t len*, *int op*);

DESCRIPTION

Restartable atomic sequences are code sequences which are guaranteed to execute without preemption. This property is assured by the kernel by re-executing a preempted sequence from the start. This functionality enables applications to build atomic sequences which, when executed to completion, will have executed atomically. Restartable atomic sequences are intended to be used on systems that do not have hardware support for low-overhead atomic primitives.

The **rasctl** function manipulates a process's set of restartable atomic sequences. If a restartable atomic sequence is registered and the process is preempted within the range *addr* and *addr*+*len*, then the process is resumed at *addr*.

As the process execution can be rolled-back, the code in the sequence should have no side effects other than a final store at *addr*+*len*−1. The kernel does not guarantee that the sequences are successfully restartable. It assumes that the application knows what it is doing. Restartable atomic sequences should adhere to the following guidelines:

- have a single entry point and a single exit point;
- not execute emulated instructions; and
- not invoke any functions or system calls.

Restartable atomic sequences are inherited from the parent by the child during the fork(2) operation. Restartable atomic sequences for a process are removed during exec(3).

The operations that can be applied to a restartable atomic sequence are specified by the *op* argument. Possible operations are:

RAS_INSTALL	Install this sequence.
RAS_PURGE	Remove the specified registered sequence for this process.
RAS_PURGE_ALL	Remove all registered sequences for this process.

The RAS_PURGE and RAS_PURGE_ALL operations should be considered to have undefined behaviour if there are any other runnable threads in the address space which might be executing within the restartable atomic sequence(s) at the time of the purge. The caller must be responsible for ensuring that there is some form of coordination with other threads to prevent unexpected behaviour.

To preserve the atomicity of sequences, the kernel attempts to protect the sequences from alteration by the ptrace(2) facility.

RETURN VALUES

Upon successful completion, **rasctl**() returns zero. Otherwise, −1 is returned and *errno* is set to indicate the error.

ERRORS

The **rasctl** function will fail if:

[EINVAL]	Invalid input was supplied, such as an invalid operation, an invalid address, or an invalid length. A process may have a finite number of atomic sequences that is defined at compile time.
[EOPNOTSUPP]	Restartable atomic sequences are not supported by the kernel.
[ESRCH]	Restartable atomic sequence not registered.

SEE ALSO

ptrace(2)

HISTORY

The **rasctl** functionality first appeared in NetBSD 2.0 based on a similar interface that appeared in Mach 2.5.

CAVEATS

Modern compilers reorder instruction sequences to optimize speed. The start address and size of a **RAS** need to be protected against this. One level of protection is created by compiler dependent instructions, abstracted from user level code via the following macros:

RAS_DECL(name)	Declares the start and end labels used internally by the other macros to mark a **RAS**. The name uniquely identifies the **RAS**.
RAS_START(name)	Marks the start of the code. Each restart returns to the instruction following this macro.
RAS_END(name)	Marks the end of the restartable code.
RAS_ADDR(name)	Returns the start address of a **RAS** and is used to create the first argument to **rasctl**.
RAS_SIZE(name)	Returns the size of a **RAS** and is used as second argument to **rasctl**.

Recent versions of gcc(1) require the **−fno-reorder-blocks** flag to prevent blocks of code wrapped with RAS_START/RAS_END being moved outside these labels. However, be aware that this may not always be sufficient to prevent gcc(1) from generating non-restartable code within the **RAS** due to register clobbers. It is, therefore, strongly recommended that restartable atomic sequences are coded in assembly. **RAS** blocks within assembly code can be specified by using the following macros:

RAS_START_ASM(name)	Similar to **RAS_START** but for use in assembly source code.
RAS_END_ASM(name)	Similar to **RAS_END** but for use in assembly source code.
RAS_START_ASM_HIDDEN(name)	Similar to **RAS_START_ASM** except that the symbol will not be placed in the dynamic symbol table.
RAS_END_ASM_HIDDEN(name)	Similar to **RAS_END_ASM** except that the symbol will not be placed in the dynamic symbol table.

NAME

read, **readv**, **pread**, **preadv** — read input

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

ssize_t
read(*int d*, *void *buf*, *size_t nbytes*);

ssize_t
pread(*int d*, *void *buf*, *size_t nbytes*, *off_t offset*);

#include <sys/uio.h>

ssize_t
readv(*int d*, *const struct iovec *iov*, *int iovcnt*);

ssize_t
preadv(*int d*, *const struct iovec *iov*, *int iovcnt*, *off_t offset*);

DESCRIPTION

read() attempts to read *nbytes* of data from the object referenced by the descriptor *d* into the buffer pointed to by *buf*. **readv**() performs the same action, but scatters the input data into the *iovcnt* buffers specified by the members of the *iov* array: iov[0], iov[1], ..., iov[iovcnt − 1]. **pread**() and **preadv**() perform the same functions, but read from the specified position in the file without modifying the file pointer.

For **readv**() and **preadv**(), the *iovec* structure is defined as:

```
struct iovec {
        void *iov_base;
        size_t iov_len;
};
```

Each *iovec* entry specifies the base address and length of an area in memory where data should be placed. **readv**() will always fill an area completely before proceeding to the next.

On objects capable of seeking, the **read**() starts at a position given by the file pointer associated with *d* (see lseek(2)). Upon return from **read**(), the file pointer is incremented by the number of bytes actually read.

Objects that are not capable of seeking always read from the current position. The value of the file pointer associated with such an object is undefined.

Upon successful completion, **read**(), **readv**(), **pread**(), and **preadv**() return the number of bytes actually read and placed in the buffer. The system guarantees to read the number of bytes requested if the descriptor references a normal file that has that many bytes left before the end-of-file, but in no other case.

RETURN VALUES

If successful, the number of bytes actually read is returned. Upon reading end-of-file, zero is returned. Otherwise, a −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

read(), **readv**(), **pread**(), and **preadv**() will succeed unless:

[EBADF]	*d* is not a valid file or socket descriptor open for reading.
[EFAULT]	*buf* points outside the allocated address space.
[EIO]	An I/O error occurred while reading from the file system.
[EINTR]	A read from a slow device (i.e. one that might block for an arbitrary amount of time) was interrupted by the delivery of a signal before any data arrived. See `sigaction`(2) for more information on the interaction between signals and system calls.
[EINVAL]	The file pointer associated with *d* was negative.
[EINVAL]	The total length of the I/O is more than can be expressed by the ssize_t return value.
[EAGAIN]	The file was marked for non-blocking I/O, and no data were ready to be read.

In addition, **readv**() and **preadv**() may return one of the following errors:

[EINVAL]	*iovcnt* was less than or equal to 0, or greater than {IOV_MAX}.
[EINVAL]	One of the *iov_len* values in the *iov* array was negative.
[EINVAL]	The sum of the *iov_len* values in the *iov* array overflowed a 32-bit integer.
[EFAULT]	Part of the *iov* points outside the process's allocated address space.

The **pread**() and **preadv**() calls may also return the following errors:

[EINVAL]	The specified file offset is invalid.
[ESPIPE]	The file descriptor is associated with a pipe, socket, or FIFO.

SEE ALSO

`dup`(2), `fcntl`(2), `open`(2), `pipe`(2), `poll`(2), `select`(2), `sigaction`(2), `socket`(2), `socketpair`(2)

STANDARDS

The **read**() function conforms to ISO/IEC 9945-1:1990 ("POSIX.1"). The **readv**() and **pread**() functions conform to X/Open Portability Guide Issue 4, Version 2 ("XPG4.2").

HISTORY

The **preadv**() function call appeared in NetBSD 1.4. The **pread**() function call appeared in AT&T System V.4 UNIX. The **readv**() function call appeared in 4.2BSD. The **read**() function call appeared in Version 6 AT&T UNIX.

CAVEATS

Error checks should explicitly test for −1. Code such as

```
while ((nr = read(fd, buf, sizeof(buf))) > 0)
```

is not maximally portable, as some platforms allow for *nbytes* to range between SSIZE_MAX and SIZE_MAX − 2, in which case the return value of an error-free **read**() may appear as a negative number distinct from −1. Proper loops should use

```
while ((nr = read(fd, buf, sizeof(buf))) != -1 && nr != 0)
```

NAME

readlink — read value of a symbolic link

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

ssize_t
readlink(*const char * restrict path*, *char * restrict buf*, *size_t bufsiz*);

DESCRIPTION

readlink() places the contents of the symbolic link *path* in the buffer *buf*, which has size *bufsiz*. **readlink**() does not append a NUL character to *buf*.

RETURN VALUES

The call returns the count of characters placed in the buffer if it succeeds, or a −1 if an error occurs, placing the error code in the global variable *errno*.

EXAMPLES

A typical use is illustrated in the following piece of code which reads the contents of a symbolic link named /symbolic/link and stores them as null-terminated string:

```
#include <limits.h>
#include <unistd.h>

char buf[PATH_MAX];
ssize_t len;

if ((len = readlink("/symbolic/link", buf, sizeof(buf)-1)) == -1)
        error handling;
buf[len] = ’\0’;
```

ERRORS

readlink() will fail if:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EINVAL]	The named file is not a symbolic link.
[EIO]	An I/O error occurred while reading from the file system.
[EFAULT]	*buf* extends outside the process’s allocated address space.

SEE ALSO

`lstat`(2), `stat`(2), `symlink`(2), `symlink`(7)

STANDARDS

The **`readlink`**() function conforms to IEEE Std 1003.1-2001 (“POSIX.1”).

HISTORY

The **`readlink`**() function appeared in 4.2BSD. The type returned was changed from *`int`* to *`ssize_t`* in NetBSD 2.1.

NAME

reboot — reboot system or halt processor

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>
#include <sys/reboot.h>

int
reboot(*int howto*, *char *bootstr*);

DESCRIPTION

reboot() reboots the system. Only the super-user may reboot a machine on demand. However, a reboot is invoked automatically in the event of unrecoverable system failures.

howto is a mask of options; the system call interface allows the following options, defined in the include file ⟨sys/reboot.h⟩, to be passed to the new kernel or the new bootstrap and init programs.

RB_AUTOBOOT	0x0000	The default, causing the system to reboot in its usual fashion.
RB_ASKNAME	0x0001	Interpreted by the bootstrap program itself, causing it to prompt on the console as to what file should be booted. Normally, the system is booted from the file "*xx*(0,0)netbsd", where *xx* is the default disk name, without prompting for the file name.
RB_DFLTROOT	0x0020	Use the compiled in root device. Normally, the system uses the device from which it was booted as the root device if possible. (The default behavior is dependent on the ability of the bootstrap program to determine the drive from which it was loaded, which is not possible on all systems.)
RB_DUMP	0x0100	Dump kernel memory before rebooting; see savecore(8) for more information.
RB_HALT	0x0008	the processor is simply halted; no reboot takes place. This option should be used with caution.
RB_POWERDOWN	0x0808	This option is always used in conjunction with RB_HALT, and if the system hardware supports the function, the system will be powered off, otherwise it has no effect.
RB_INITNAME	0x0010	An option allowing the specification of an init program (see init(8)) other than /sbin/init to be run when the system reboots. This switch is not currently available.
RB_KDB	0x0040	Load the symbol table and enable a built-in debugger in the system. This option will have no useful function if the kernel is not configured for debugging. Several other options have different meaning if combined with this option, although their use may not be possible via the **reboot**() call. See ddb(4) for more information.
RB_NOSYNC	0x0004	Normally, the disks are sync'd (see sync(8)) before the processor is halted or rebooted. This option may be useful if file system changes have been made manually or if the processor is on fire.
RB_RDONLY	0x0080	Initially mount the root file system read-only. This is currently the default, and this option has been deprecated.
RB_SINGLE	0x0002	Normally, the reboot procedure involves an automatic disk consistency check and then multi-user operations. RB_SINGLE prevents this, booting the system with a single-user shell on the console. RB_SINGLE is actually interpreted by the init(8) program in the newly booted system.

		When no options are given (i.e., RB_AUTOBOOT is used), the system is rebooted from file ‘‘netbsd’’ in the root file system of unit 0 of a disk chosen in a processor specific way. An automatic consistency check of the disks is normally performed (see fsck(8)).
RB_STRING	0x0400	*bootstr* is a string passed to the firmware on the machine, if possible, if this option is set. Currently this is only implemented on the sparc and the sun3 ports.
RB_USERCONF	0x1000	Initially invoke the userconf(4) facility when the system starts up again, if it has been compiled into the kernel that is loaded.

RETURN VALUES

If successful, this call never returns. Otherwise, a −1 is returned and an error is returned in the global variable *errno*.

ERRORS

[EPERM]	The caller is not the super-user.

SEE ALSO

ddb(4), crash(8), halt(8), init(8), reboot(8), savecore(8)

HISTORY

The **reboot**() function call appeared in 4.0BSD.

The RB_DFLTROOT option is now *obsolete*.

NAME

recv, **recvfrom**, **recvmsg** — receive a message from a socket

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/socket.h>

ssize_t
recv(*int s*, *void *buf*, *size_t len*, *int flags*);

ssize_t
recvfrom(*int s*, *void * restrict buf*, *size_t len*, *int flags*,
    *struct sockaddr * restrict from*, *socklen_t * restrict fromlen*);

ssize_t
recvmsg(*int s*, *struct msghdr *msg*, *int flags*);

DESCRIPTION

recvfrom() and **recvmsg**() are used to receive messages from a socket, and may be used to receive data on a socket whether or not it is connection-oriented.

If *from* is non-nil, and the socket is not connection-oriented, the source address of the message is filled in. *fromlen* is a value-result parameter, initialized to the size of the buffer associated with *from*, and modified on return to indicate the actual size of the address stored there.

The **recv**() call is normally used only on a *connected* socket (see connect(2)) and is identical to **recvfrom**() with a nil *from* parameter. As it is redundant, it may not be supported in future releases.

All three routines return the length of the message on successful completion. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see socket(2)).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking (see fcntl(2)) in which case the value −1 is returned and the external variable *errno* set to EAGAIN. The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested; this behavior is affected by the socket-level options SO_RCVLOWAT and SO_RCVTIMEO described in getsockopt(2).

The select(2) or poll(2) call may be used to determine when more data arrive.

The *flags* argument to a recv call is formed by *or*'ing one or more of the values:

MSG_OOB	process out-of-band data
MSG_PEEK	peek at incoming message
MSG_WAITALL	wait for full request or error

The MSG_OOB flag requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at the head of the normal data queue, and thus this flag cannot be used with such protocols. The MSG_PEEK flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data. The MSG_WAITALL flag requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if a signal is caught, an error or disconnect occurs, or the next data to be received is of a different type than that returned.

The **recvmsg**() call uses a *msghdr* structure to minimize the number of directly supplied parameters. This structure has the following form, as defined in ⟨sys/socket.h⟩:

```
struct msghdr {
        void           *msg_name;     /* optional address */
        socklen_t      msg_namelen;   /* size of address */
        struct iovec   *msg_iov;      /* scatter/gather array */
        int            msg_iovlen;    /* # elements in msg_iov */
        void           *msg_control;  /* ancillary data, see below */
        socklen_t      msg_controllen; /* ancillary data buffer len */
        int            msg_flags;     /* flags on received message */
};
```

Here *msg_name* and *msg_namelen* specify the source address if the socket is unconnected; *msg_name* may be given as a null pointer if no names are desired or required. If the socket is connected, *msg_name* and *msg_namelen* are ignored. *msg_iov* and *msg_iovlen* describe scatter gather locations, as discussed in read(2). *msg_control*, which has length *msg_controllen*, points to a buffer for other protocol control related messages or other miscellaneous ancillary data. The messages are of the form:

```
struct cmsghdr {
        socklen_t      cmsg_len;      /* data byte count, including hdr */
        int            cmsg_level;    /* originating protocol */
        int            cmsg_type;     /* protocol-specific type */
/* followed by
        u_char         cmsg_data[]; */
};
```

As an example, one could use this to learn of changes in the data-stream in XNS/SPP, or in ISO, to obtain user-connection-request data by requesting a recvmsg with no data buffer provided immediately after an **accept**() call.

Open file descriptors are now passed as ancillary data for AF_LOCAL domain sockets, with *cmsg_level* set to SOL_SOCKET and *cmsg_type* set to SCM_RIGHTS.

The *msg_flags* field is set on return according to the message received. MSG_EOR indicates end-of-record; the data returned completed a record (generally used with sockets of type SOCK_SEQPACKET). MSG_TRUNC indicates that the trailing portion of a datagram was discarded because the datagram was larger than the buffer supplied. MSG_CTRUNC indicates that some control data were discarded due to lack of space in the buffer for ancillary data. MSG_OOB is returned to indicate that expedited or out-of-band data were received.

RETURN VALUES

These calls return the number of bytes received, or −1 if an error occurred.

ERRORS

The calls fail if:

[EBADF]	The argument *s* is an invalid descriptor.
[ENOTCONN]	The socket is associated with a connection-oriented protocol and has not been connected (see connect(2) and accept(2)).
[ENOTSOCK]	The argument *s* does not refer to a socket.
[EAGAIN]	The socket is marked non-blocking, and the receive operation would block, or a receive timeout had been set, and the timeout expired before data were received.
[EINTR]	The receive was interrupted by delivery of a signal before any data were available.

[EFAULT]	The receive buffer pointer(s) point outside the process's address space.
[EINVAL]	The total length of the I/O is more than can be expressed by the ssize_t return value.

recvmsg() will also fail if:

[EMSGSIZE]	The *msg_iovlen* member of the *msg* structure is less than or equal to 0 or is greater than {IOV_MAX}.

SEE ALSO

fcntl(2), getsockopt(2), poll(2), read(2), select(2), socket(2)

HISTORY

The **recv**() function call appeared in 4.2BSD.

NAME

rename — change the name of a file

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <stdio.h>

int
rename(*const char ∗from*, *const char ∗to*);

DESCRIPTION

rename() causes the link named *from* to be renamed as *to*. If *to* exists, it is first removed. Both *from* and *to* must be of the same type (that is, both directories or both non-directories), and must reside on the same file system.

rename() guarantees that an instance of *to* will always exist, even if the system should crash in the middle of the operation.

If the final component of *from* is a symbolic link, the symbolic link is renamed, not the file or directory to which it points.

If both *from* and *to* are pathnames of the same existing file in the file system's name space, **rename**() returns successfully and performs no other action.

RETURN VALUES

A 0 value is returned if the operation succeeds, otherwise **rename**() returns −1 and the global variable *errno* indicates the reason for the failure.

ERRORS

rename() will fail and neither of the argument files will be affected if:

[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	A component of the *from* path does not exist, or a path prefix of *to* does not exist.
[EACCES]	A component of either path prefix denies search permission, or the requested link requires writing in a directory with a mode that denies write permission.
[EPERM]	The directory containing *from* is marked sticky, and neither the containing directory nor *from* are owned by the effective user ID. Or the *to* file exists, the directory containing *to* is marked sticky, and neither the containing directory nor *to* are owned by the effective user ID.
[ELOOP]	Too many symbolic links were encountered in translating either pathname.
[ENOTDIR]	A component of either path prefix is not a directory, or *from* is a directory, but *to* is not a directory.
[EISDIR]	*to* is a directory, but *from* is not a directory.
[EXDEV]	The link named by *to* and the file named by *from* are on different logical devices (file systems). Note that this error code will not be returned if the implementation permits cross-device links.

`[ENOSPC]`	The directory in which the entry for the new name is being placed cannot be extended because there is no space left on the file system containing the directory.
`[EDQUOT]`	The directory in which the entry for the new name is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
`[EIO]`	An I/O error occurred while making or updating a directory entry.
`[EROFS]`	The requested link requires writing in a directory on a read-only file system.
`[EFAULT]`	*Path* points outside the process's allocated address space.
`[EINVAL]`	*`from`* is a parent directory of *`to`*, or an attempt is made to rename '`.`' or '`..`'.
`[ENOTEMPTY]`	*`to`* is a directory and is not empty.
`[EBUSY]`	*`from`* or *`to`* is the mount point for a mounted file system.

SEE ALSO

`open`(2), `symlink`(7)

STANDARDS

The **`rename`**() function deviates from the semantics defined in ISO/IEC 9945-1:1990 ("POSIX.1"), which specifies that if both *`from`* and *`to`* *link* to the same existing file, **`rename`**() shall return successfully and performs no further action, whereas this implementation will remove the file specified by *`from`* unless both *`from`* and *`to`* are pathnames of the same file in the file system's name space.

To retain conformance, a compatibility interface is provided by the POSIX Compatibility Library (libposix, −lposix) which is also be brought into scope if any of the `_POSIX_SOURCE`, `_POSIX_C_SOURCE` or `_XOPEN_SOURCE` preprocessor symbols are defined at compile-time: the **`rename`**() function conforms to ISO/IEC 9945-1:1990 ("POSIX.1") and X/Open Portability Guide Issue 4, Version 2 ("XPG4.2").

BUGS

The system can deadlock if a loop in the file system graph is present. This loop takes the form of an entry in directory `a`, say `a/foo`, being a hard link to directory `b`, and an entry in directory `b`, say `b/bar`, being a hard link to directory `a`. When such a loop exists and two separate processes attempt to perform `rename a/foo b/bar` and `rename b/bar a/foo`, respectively, the system may deadlock attempting to lock both directories for modification. Hard links to directories should be replaced by symbolic links by the system administrator.

NAME

revoke — revoke file access

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
revoke(*const char *path*);

DESCRIPTION

The **revoke** function invalidates all current open file descriptors in the system for the file named by *path*. Subsequent operations on any such descriptors fail, with the exceptions that a **read**() from a character device file which has been revoked returns a count of zero (end of file), and a **close**() call will succeed. If the file is a special file for a device which is open, the device close function is called as if all open references to the file had been closed.

Access to a file may be revoked only by its owner or the super user.

revoke is normally used to prepare a terminal device for a new login session, preventing any access by a previous user of the terminal.

RETURN VALUES

A 0 value indicates that the call succeeded. A −1 return value indicates an error occurred and *errno* is set to indicate the reason.

ERRORS

Access to the named file is revoked unless one of the following:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire path name exceeded 1024 characters.
[ENOENT]	The named file or a component of the path name does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EFAULT]	*path* points outside the process's allocated address space.
[EPERM]	The caller is neither the owner of the file nor the super user.

SEE ALSO

close(2), dup(2), fcntl(2), flock(2), fstat(2), read(2), write(2)

HISTORY

The **revoke** function was introduced in 4.3BSD−Reno.

NAME

rmdir — remove a directory file

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
rmdir(*const char *path*);

DESCRIPTION

rmdir() removes a directory file whose name is given by *path*. The directory must not have any entries other than '.' and '..'.

RETURN VALUES

A 0 is returned if the remove succeeds; otherwise a −1 is returned and an error code is stored in the global location *errno*.

ERRORS

The named file is removed unless:

[ENOTDIR]	A component of the path is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	The named directory does not exist.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[ENOTEMPTY]	The named directory contains files other than '.' and '..' in it.
[EACCES]	Search permission is denied for a component of the path prefix, or write permission is denied on the directory containing the link to be removed.
[EPERM]	The directory containing the directory to be removed is marked sticky, and neither the containing directory nor the directory to be removed are owned by the effective user ID.
[EBUSY]	The directory to be removed is the mount point for a mounted file system.
[EIO]	An I/O error occurred while deleting the directory entry or deallocating the inode.
[EROFS]	The directory entry to be removed resides on a read-only file system.
[EFAULT]	*path* points outside the process's allocated address space.

SEE ALSO

mkdir(2), unlink(2)

STANDARDS

The **rmdir**() function conforms to ISO/IEC 9945-1:1990 ("POSIX.1").

HISTORY

The **`rmdir`**() function call appeared in 4.2BSD.

NAME

sa_enable — enable scheduler activation

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sa.h>

int
sa_enable();

DESCRIPTION

sa_enable() is used to enable scheduler activation.

An upcall handler and upcall stacks should be registered with **sa_register**() and **sa_stacks**() beforehand.

RETURN VALUES

On success, **sa_enable**() will not return to userland in the normal way. It returns into the upcall hander with an SA_UPCALL_NEWPROC upcall. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

sa_register(2), sa_stacks(2), pthread(3)

NAME

sa_register — register a scheduler activation upcall handler

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sa.h>

int
sa_register(*sa_upcall_t new*, *sa_upcall_t *old*, *int flags*,
    ssize_t stackinfo_offset);

DESCRIPTION

sa_register is used to prepare scheduler activation. It registers the upcall handler specified by *new*. If *old* isn't NULL, the previous handler will be returned to the location pointed by it. If SA_FLAG_STACKINFO is specified in *flags*, *stackinfo_offset* is used to locate per upcall stack memory areas shared between userland and kernel. *stackinfo_offset* is a byte offset from the corresponding upcall stack.

RETURN VALUES

sa_register() returns zero on success. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

pthread(3)

NAME

sa_setconcurrency — increase the number of virtual processors

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sa.h>

int
sa_setconcurrency(*int concurrency*);

DESCRIPTION

sa_setconcurrency() is used to increase the number of scheduler activation virtual processors used by the process. **sa_setconcurrency**() increases the number of virtual processors, i.e., execution concurrency, up to *concurrency*. However, **sa_setconcurrency**() won't try to allocate more virtual processors than there are physical processors on the system.

Scheduler activation should be enabled by **sa_enable**() beforehand.

RETURN VALUES

On success, **sa_setconcurrency**() returns the number of newly added virtual processors. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

pthread(3)

NAME

sa_stacks — register scheduler activation upcall stacks

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sa.h>

int
sa_stacks(*int num*, *stack_t ∗stacks*);

DESCRIPTION

sa_stacks() is used to register scheduler activation upcall stacks. *stacks* is an array of *num* of *stack_t*.

An upcall handler should be registered by **sa_register**() beforehand.

RETURN VALUES

On success, **sa_stacks**() returns the number of stacks registered. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

sa_register(2), pthread(3)

NAME

sa_yield — idle a virtual processor

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sa.h>

int
sa_yield();

DESCRIPTION

sa_yield() is used to idle a virtual processor on which the calling thread is running. It's typically used when the process's userland scheduler has no userland thread to run.

Scheduler activation should be enabled by **sa_enable**() beforehand.

RETURN VALUES

On success, **sa_yield**() will not return to userland in the normal way. It returns into an upcall hander with an upcall. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

sa_enable(2), pthread(3)

NAME

select, **pselect** — synchronous I/O multiplexing

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/select.h>

int
select(*int nfds*, *fd_set * restrict readfds*, *fd_set * restrict writefds*,
    *fd_set * restrict exceptfds*, *struct timeval * restrict timeout*);

int
pselect(*int nfds*, *fd_set * restrict readfds*, *fd_set * restrict writefds*,
    *fd_set * restrict exceptfds*, *const struct timespec *restrict timeout*,
    *const sigset_t * restrict sigmask*);

FD_SET(*int fd*, *fd_set *fdset*);

FD_CLR(*int fd*, *fd_set *fdset*);

FD_ISSET(*int fd*, *fd_set *fdset*);

FD_ZERO(*fd_set *fdset*);

DESCRIPTION

select() and **pselect**() examine the I/O descriptor sets whose addresses are passed in *readfds*, *writefds*, and *exceptfds* to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first *nfds* descriptors are checked in each set; i.e., the descriptors from 0 through *nfds*−1 in the descriptor sets are examined. This means that *nfds* must be set to the highest file descriptor of the three sets, plus one. On return, **select**() and **pselect**() replace the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. **select**() and **pselect**() return the total number of ready descriptors in all the sets.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets: **FD_ZERO**(*fdset*) initializes a descriptor set pointed to by *fdset* to the null set. **FD_SET**(*fd*, *fdset*) includes a particular descriptor *fd* in *fdset*. **FD_CLR**(*fd*, *fdset*) removes *fd* from *fdset*. **FD_ISSET**(*fd*, *fdset*) is non-zero if *fd* is a member of *fdset*, zero otherwise. The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to FD_SETSIZE, which is normally at least equal to the maximum number of descriptors supported by the system.

If *timeout* is a non-null pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a null pointer, the select blocks indefinitely. To affect a poll, the *timeout* argument should be non-null, pointing to a zero-valued timeval or timespec structure, as appropriate. *timeout* is not changed by **select**(), and may be reused on subsequent calls; however, it is good style to re-initialize it before each invocation of **select**().

If *sigmask* is a non-null pointer, then the **pselect**() function shall replace the signal mask of the caller by the set of signals pointed to by *sigmask* before examining the descriptors, and shall restore the signal mask of the calling thread before returning.

Any of *readfds*, *writefds*, and *exceptfds* may be given as null pointers if no descriptors are of interest.

RETURN VALUES

select() returns the number of ready descriptors that are contained in the descriptor sets, or −1 if an error occurred. If the time limit expires, **select**() returns 0. If **select**() returns with an error, including one due to an interrupted call, the descriptor sets will be unmodified.

EXAMPLES

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <err.h>
#include <errno.h>
#include <sysexits.h>
#include <sys/types.h>
#include <sys/time.h>

int
main(argc, argv)
        int argc;
        char **argv;
{
        fd_set read_set;
        struct timeval timeout;
        int ret, fd, i;

        /* file descriptor 1 is stdout */
        fd = 1;

        /* Wait for ten seconds. */
        timeout.tv_sec = 10;
        timeout.tv_usec = 0;

        /* Initialize the read set to null */
        FD_ZERO(&read_set);

        /* Add file descriptor 1 to read_set */
        FD_SET(fd, &read_set);

        /*
         * Check if data is ready to be readen on
         * file descriptor 1, give up after 10 seconds.
         */
        ret = select(fd + 1, &read_set, NULL, NULL, &timeout);

        /*
         * Returned value is the number of file
         * descriptors ready for I/O, or -1 on error.
         */
        switch (ret) {
        case −1:
                err(EX_OSERR, "select() failed");
                break;
        case 0:
                printf("Timeout, no data received.\n");
                break;

        default:
                printf("Data received on %d file desciptor(s)\n", ret);

                /*
                 * select(2) hands back a file descriptor set where
                 * only descriptors ready for I/O are set. These can
                 * be tested using FD_ISSET
                 */
                for (i = 0; i <= fd; i++) {
                        if (FD_ISSET(i, &read_set)) {
                                printf("Data on file descriptor %d\n", i);
                                /* Remove the file descriptor from the set */
                                FD_CLR(fd, &read_set);
                        }
                }
                break;
        }

        return 0;
}
```

ERRORS

An error return from **select**() indicates:

[EBADF]	One of the descriptor sets specified an invalid descriptor.
[EFAULT]	One or more of *readfds*, *writefds*, or *exceptfds* points outside the process's allocated address space.
[EINTR]	A signal was delivered before the time limit expired and before any of the selected events occurred.
[EINVAL]	The specified time limit is invalid. One of its components is negative or too large.

SEE ALSO

accept(2), connect(2), gettimeofday(2), poll(2), read(2), recv(2), send(2), write(2), getdtablesize(3)

HISTORY

The **select**() function call appeared in 4.2BSD.

BUGS

Although the provision of getdtablesize(3) was intended to allow user programs to be written independent of the kernel limit on the number of open files, the dimension of a sufficiently large bit field for select remains a problem. The default bit size of *fd_set* is based on the symbol FD_SETSIZE (currently 256), but that is somewhat smaller than the current kernel limit to the number of open files. However, in order to accommodate programs which might potentially use a larger number of open files with select, it is possible to increase this size within a program by providing a larger definition of FD_SETSIZE before the inclusion of ⟨sys/types.h⟩. The kernel will cope, and the userland libraries provided with the system are also ready for large numbers of file descriptors.

Note: `rpc`(3) library uses *fd_set* with the default `FD_SETSIZE` as part of its ABI. Therefore, programs that use `rpc`(3) routines cannot change `FD_SETSIZE`.

Alternatively, to be really safe, it is possible to allocate *fd_set* bit-arrays dynamically. The idea is to permit a program to work properly even if it is `execve`(2)'d with 4000 file descriptors pre-allocated. The following illustrates the technique which is used by userland libraries:

```
fd_set *fdsr;
int max = fd;

fdsr = (fd_set *)calloc(howmany(max+1, NFDBITS),
    sizeof(fd_mask));
if (fdsr == NULL) {
        ...
        return (-1);
}
FD_SET(fd, fdsr);
n = select(max+1, fdsr, NULL, NULL, &tv);
...
free(fdsr);
```

Alternatively, it is possible to use the `poll`(2) interface. `poll`(2) is more efficient when the size of **select**()'s *fd_set* bit-arrays are very large, and for fixed numbers of file descriptors one need not size and dynamically allocate a memory object.

select() should probably have been designed to return the time remaining from the original timeout, if any, by modifying the time value in place. Even though some systems stupidly act in this different way, it is unlikely this semantic will ever be commonly implemented, as the change causes massive source code compatibility problems. Furthermore, recent new standards have dictated the current behaviour. In general, due to the existence of those non-conforming systems, it is unwise to assume that the timeout value will be unmodified by the **select**() call, and the caller should reinitialize it on each invocation. Calculating the delta is easily done by calling `gettimeofday`(2) before and after the call to **select**(), and using **timersub**() (as described in `getitimer`(2)).

Internally to the kernel, **select**() works poorly if multiple processes wait on the same file descriptor.

NAME

semctl — semaphore control operations

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/sem.h>

int
semctl(*int semid*, *int semnum*, *int cmd*, *...*);

DESCRIPTION

The **semctl**() system call provides a number of control operations on the semaphore specified by *semnum* and *semid*. The operation to be performed is specified in *cmd* (see below). The fourth argument is optional and depends upon the operation requested. If required, it is a union of the following fields:

```
int      val;            /* value for SETVAL */
struct   semid_ds *buf;  /* buffer for IPC_{STAT,SET} */
u_short *array;          /* array for GETALL & SETALL */
```

The **semid_ds** structure used in the IPC_SET and IPC_STAT commands is defined in ⟨sys/sem.h⟩ and contains the following members:

```
struct ipc_perm sem_perm; /* operation permissions */
unsigned short sem_nsems; /* number of sems in set */
time_t sem_otime;         /* last operation time */
time_t sem_ctime;         /* last change time */
```

The **ipc_perm** structure used inside the **semid_ds** structure is defined in ⟨sys/ipc.h⟩ and contains the following members:

```
uid_t cuid;  /* creator user id */
gid_t cgid;  /* creator group id */
uid_t uid;  /* user id */
gid_t gid;  /* group id */
mode_t mode; /* permission (lower 9 bits) */
```

semctl() provides the following operations:

GETVAL	Return the value of the semaphore.
SETVAL	Set the value of the semaphore to *arg.val*, where *arg* is the fourth argument to **semctl**().
GETPID	Return the pid of the last process that did an operation on this semaphore.
GETNCNT	Return the number of processes waiting to acquire the semaphore.
GETZCNT	Return the number of processes waiting for the value of the semaphore to reach 0.
GETALL	Return the values of all the semaphores associated with *semid*.
SETALL	Set the values of all the semaphores that are associated with the semaphore identifier *semid* to the corresponding values in *arg.array*, where *arg* is the fourth argument to **semctl**().
IPC_STAT	Gather information about a semaphore and place the information in the structure pointed to by *arg.buf*, where *arg* is the fourth argument to **semctl**().

`IPC_SET` Set the value of the *sem_perm.uid*, *sem_perm.gid* and *sem_perm.mode* fields in the structure associated with the semaphore. The values are taken from the corresponding fields in the structure pointed to by *`arg.buf`*, there *`arg`* is the fourth argument to **`semctl`**(). This operation can only be executed by the super-user, or a process that has an effective user id equal to either *sem_perm.cuid* or *sem_perm.uid* in the data structure associated with the message queue.

`IPC_RMID` Remove the semaphores associated with *`semid`* from the system and destroy the data structures associated with it. Only the super-user or a process with an effective uid equal to the *sem_perm.cuid* or *sem_perm.uid* values in the data structure associated with the semaphore can do this.

The permission to read or change a message queue (see `semop`(2)) is determined by the *sem_perm.mode* field in the same way as is done with files (see `chmod`(2)), but the effective uid can match either the *sem_perm.cuid* field or the *sem_perm.uid* field, and the effective gid can match either *sem_perm.cgid* or *sem_perm.gid*.

RETURN VALUES

For the `GETVAL`, `GETPID`, `GETNCNT`, and `GETZCNT` operations, **`semctl`**() returns one of the values described above if successful. All other operations will make **`semctl`**() return 0 if no errors occur. Otherwise −1 is returned and *errno* set to reflect the error.

ERRORS

`semctl`() will fail if:

`[EPERM]` *`cmd`* is equal to `IPC_SET` or `IPC_RMID` and the caller is not the super-user, nor does the effective uid match either the *sem_perm.uid* or *sem_perm.cuid* fields of the data structure associated with the message queue.

`[EACCES]` The caller has no operation permission for this semaphore.

`[EINVAL]` *`semid`* is not a valid message semaphore identifier.

cmd is not a valid command.

`[EFAULT]` *`arg.buf`* or *`arg.array`* specifies an invalid address.

`[ERANGE]` *`cmd`* is equal to `SETVAL` or `SETALL` and the value to be set is greater than the system semaphore maximum value.

SEE ALSO

`semget`(2), `semop`(2)

STANDARDS

The **`semctl`** system call conforms to X/Open System Interfaces and Headers Issue 5 (“XSH5”).

HISTORY

Semaphores appeared in the first release of AT&T System V UNIX.

NAME

semget — get set of semaphores

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/sem.h>

int
semget(*key_t key*, *int nsems*, *int semflg*);

DESCRIPTION

The **semget**() system call returns the semaphore identifier associated with *key*.

A new set containing *nsems* semaphores is created if either *key* is equal to IPC_PRIVATE, or *key* does not have a semaphore set associated with it and the IPC_CREAT bit is set in *semflg*. If both the IPC_CREAT bit and the IPC_EXCL bit are set in *semflg*, and *key* has a semaphore set associated with it already, the operation will fail.

If a new set of semaphores is created, the data structure associated with it (the *semid_ds* structure, see semctl(2)) is initialized as follows:

- *sem_perm.cuid* and *sem_perm.uid* are set to the effective uid of the calling process.
- *sem_perm.gid* and *sem_perm.cgid* are set to the effective gid of the calling process.
- *sem_perm.mode* is set to the lower 9 bits of *semflg*.
- *sem_nsems* is set to the value of *nsems*.
- *sem_ctime* is set to the current time.
- *sem_otime* is set to 0.

RETURN VALUES

semget() returns a non-negative semaphore identifier if successful. Otherwise, −1 is returned and *errno* is set to reflect the error.

ERRORS

[EACCES]	The caller has no permission to access a semaphore set already associated with *key*.
[EEXIST]	Both IPC_CREAT and IPC_EXCL are set in *semflg*, and a semaphore set is already associated with *key*.
[EINVAL]	*nsems* is less than 0 or greater than the system limit for the number in a semaphore set.
	A semaphore set associated with *key* exists, but has fewer semaphores than the number specified in *nsems*.
[ENOSPC]	A new set of semaphores could not be created because the system limit for the number of semaphores or the number of semaphore sets has been reached.
[ENOENT]	IPC_CREAT is not set in *semflg* and no semaphore set associated with *key* was found.

SEE ALSO

`ipcs`(1), `semctl`(2), `semop`(2), `ftok`(3)

STANDARDS

The **`semget`** system call conforms to X/Open System Interfaces and Headers Issue 5 (“XSH5”).

HISTORY

Semaphores appeared in the first release of AT&T System V UNIX.

NAME

semop — semaphore operations

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/sem.h>

int
semop(*int semid*, *struct sembuf *sops*, *size_t nsops*);

DESCRIPTION

semop() provides a number of atomic operations on a set of semaphores. The semaphore set is specified by *semid*, *sops* is an array of semaphore operations, and *nsops* is the number of operations in this array. The *sembuf* structures in the array contain the following members:

```
unsigned short sem_num; /* semaphore # */
short          sem_op;  /* semaphore operation */
short          sem_flg; /* operation flags */
```

Each operation (specified in *sem_op*) is applied to semaphore number *sem_num* in the set of semaphores specified by *semid*. The value of *sem_op* determines the action taken in the following way:

- *sem_op* is less than 0. The current process is blocked until the value of the semaphore is greater than or equal to the absolute value of *sem_op*. The absolute value of *sem_op* is then subtracted from the value of the semaphore, and the calling process continues. Negative values of *sem_op* are thus used to enter critical regions.
- *sem_op* is greater than 0. Its value is added to the value of the specified semaphore. This is used to leave critical regions.
- *sem_op* is equal to 0. The calling process is blocked until the value of the specified semaphore reaches 0.

The behaviour of each operation is influenced by the flags set in *sem_flg* in the following way:

IPC_NOWAIT	In the case where the calling process would normally block, waiting for a semaphore to reach a certain value, IPC_NOWAIT makes the call return immediately, returning a value of −1 and setting *errno* to EAGAIN.
SEM_UNDO	Keep track of the changes that this call makes to the value of a semaphore, so that they can be undone when the calling process terminates. This is useful to prevent other processes waiting on a semaphore to block forever, should the process that has the semaphore locked terminate in a critical section.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

semop() will fail if:

[EINVAL]	There is no semaphore associated with *semid*.
[EIDRM]	The semaphore set was removed while the process was waiting for one of its semaphores to reach a certain value.

[EACCES]	The calling process has no permission to access the specified semaphore set.
[E2BIG]	The value of *nsops* is too big. The maximum is defined as MAX_SOPS in ⟨sys/sem.h⟩.
[EFBIG]	*sem_num* in one of the sem_buf structures is less than 0, or greater than the actual number of semaphores in the set specified by *semid*.
[ENOSPC]	SEM_UNDO was requested, and there is not enough space left in the kernel to store the undo information.
[EAGAIN]	The requested operation can not immediately be performed, and IPC_NOWAIT was set in *sem_flg*.
[EFAULT]	*sops* points to an illegal address.

SEE ALSO

semctl(2), semget(2)

STANDARDS

The **semop** system call conforms to X/Open System Interfaces and Headers Issue 5 (“XSH5”).

HISTORY

Semaphores appeared in the first release of AT&T System V UNIX.

NAME

send, **sendto**, **sendmsg** — send a message from a socket

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/socket.h>

ssize_t
send(*int s*, *const void *msg*, *size_t len*, *int flags*);

ssize_t
sendto(*int s*, *const void *msg*, *size_t len*, *int flags*,
*const struct sockaddr *to*, *socklen_t tolen*);

ssize_t
sendmsg(*int s*, *const struct msghdr *msg*, *int flags*);

DESCRIPTION

send(), **sendto**(), and **sendmsg**() are used to transmit a message to another socket. **send**() may be used only when the socket is in a *connected* state, while **sendto**() and **sendmsg**() may be used at any time.

The address of the target is given by *to* with *tolen* specifying its size. The length of the message is given by *len*. If the message is too long to pass atomically through the underlying protocol, the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a **send**(). Locally detected errors are indicated by a return value of −1.

If no messages space is available at the socket to hold the message to be transmitted, then **send**() normally blocks, unless the socket has been placed in non-blocking I/O mode. The select(2) or poll(2) call may be used to determine when it is possible to send more data.

The *flags* parameter may include one or more of the following:

```
#define MSG_OOB        0x0001 /* process out-of-band data */
#define MSG_PEEK       0x0002 /* peek at incoming message */
#define MSG_DONTROUTE  0x0004 /* bypass routing, use direct interface */
#define MSG_EOR        0x0008 /* data completes record */
#define MSG_NOSIGNAL   0x0400 /* do not generate SIGPIPE on EOF */
```

The flag MSG_OOB is used to send “out-of-band” data on sockets that support this notion (e.g. SOCK_STREAM); the underlying protocol must also support “out-of-band” data. MSG_EOR is used to indicate a record mark for protocols which support the concept. MSG_DONTROUTE is usually used only by diagnostic or routing programs.

See recv(2) for a description of the *msghdr* structure. MSG_NOSIGNAL is used to prevent SIGPIPE generation when writing a socket that may be closed.

RETURN VALUES

The call returns the number of characters sent, or −1 if an error occurred.

ERRORS

send(), **sendto**(), and **sendmsg**() fail if:

`[EBADF]`	An invalid descriptor was specified.
`[ENOTSOCK]`	The argument *s* is not a socket.
`[EFAULT]`	An invalid user space address was specified for a parameter.
`[EMSGSIZE]`	The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.
`[EPIPE]`	In a connected socket the connection has been broken.
`[EDSTADDRREQ]`	In a non-connected socket a destination address has not been specified.
`[EAGAIN|EWOULDBLOCK]`	The socket is marked non-blocking and the requested operation would block.
`[ENOBUFS]`	The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.
`[ENOBUFS]`	The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.
`[EACCES]`	The SO_BROADCAST option is not set on the socket, and a broadcast address was given as the destination.
`[EHOSTUNREACH]`	The destination for the message is unreachable.
`[EHOSTDOWN]`	The destination is a host on the local subnet and does not respond to `arp`(4).
`[EINVAL]`	The total length of the I/O is more than can be expressed by the ssize_t return value.
`[EAFNOSUPPORT]`	Addresses in the specified address family cannot be used with this socket.

sendto() will also fail if:

`[EISCONN]`	A destination address was specified and the socket is already connected.

sendmsg() will also fail if:

`[EMSGSIZE]`	The *msg_iovlen* member of the *msg* structure is less than or equal to 0 or is greater than `{IOV_MAX}`.

SEE ALSO

`fcntl`(2), `getsockopt`(2), `recv`(2), `select`(2), `socket`(2), `write`(2)

HISTORY

The **send**() function call appeared in 4.2BSD.

NAME

setgroups — set group access list

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/param.h>
#include <unistd.h>

int
setgroups(*int ngroups*, *const gid_t *gidset*);

DESCRIPTION

setgroups() sets the group access list of the current user process according to the array *gidset*. The parameter *ngroups* indicates the number of entries in the array and must be no more than {NGROUPS_MAX}.

Only the super-user may set new groups.

This system call affects only secondary groups.

RETURN VALUES

A 0 value is returned on success, −1 on error, with an error code stored in *errno*.

ERRORS

The **setgroups**() call will fail if:

[EINVAL]	The value of *ngroups* is greater than {NGROUPS_MAX}.
[EPERM]	The caller is not the super-user.
[EFAULT]	The address specified for *gidset* is outside the process address space.

SEE ALSO

getgroups(2), setegid(2), setgid(2), setregid(2), initgroups(3)

HISTORY

The **setgroups**() function call appeared in 4.2BSD.

NAME

setpgid, **setpgrp** — set process group

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
setpgid(*pid_t pid*, *pid_t pgrp*);

int
setpgrp(*pid_t pid*, *pid_t pgrp*);

DESCRIPTION

setpgid() sets the process group of the specified process *pid* to the specified *pgrp*. If *pid* is zero, then the call applies to the current process. If *pgrp* is zero, then the process id of the process specified by *pid* is used instead.

If the invoker is not the super-user, then the affected process must have the same effective user-id as the invoker or be a descendant of the invoking process.

RETURN VALUES

setpgid() returns 0 when the operation was successful. If the request failed, −1 is returned and the global variable *errno* indicates the reason.

ERRORS

setpgid() will fail and the process group will not be altered if:

[EACCES]	The value of the *pid* argument matches the process ID of a child process of the calling process, and the child process has successfully executed one of the exec functions.
[EINVAL]	The value of the *pgrp* argument is less than zero.
[EPERM]	The effective user ID of the requested process is different from that of the caller and the process is not a descendant of the calling process.
[ESRCH]	The value of the *pid* argument does not match the process ID of the calling process or of a child process of the calling process.

SEE ALSO

getpgrp(2)

STANDARDS

The **setpgid**() function conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

COMPATIBILITY

setpgrp() is identical to **setpgid**(), and is retained for calling convention compatibility with historical versions of BSD.

NAME

setregid — set real and effective group ID's

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
setregid(*gid_t rgid*, *gid_t egid*);

DESCRIPTION

This interface is made obsolete by the saved ID functionality in setgid(2) and setegid(2).

The real and effective group ID's of the current process are set according to the arguments. If the real group ID is changed, the saved group ID is changed to the new value of the effective group ID.

If *rgid* or *egid* is −1, the current gid is filled in by the system. Unprivileged users may change the real group ID to the effective group ID, and may change the effective group ID to the real group ID or the saved group ID; only the super-user may make other changes.

The **setregid**() function has been used to swap the real and effective group IDs in set-group-ID programs to temporarily relinquish the set-group-ID value. This purpose is now better served by the use of the **setegid**() function (see setgid(2)).

When setting the real and effective group IDs to the same value, this function is equivalent to the **setgid**() function. When setting only the effective group ID, this function is equivalent to the **setegid**() function.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

[EPERM]	The current process is not the super-user and a change other than changing the effective group-id to the real group-id was specified.

SEE ALSO

getgid(2), setegid(2), setgid(2), setuid(2)

HISTORY

The **setregid**() function call appeared in 4.2BSD. An incompatible version was implemented in 4.4BSD. It was reimplemented in NetBSD 1.2 in a way compatible with 4.3BSD, SunOS and Linux, but should not be used in new code.

NAME

setreuid — set real and effective user ID's

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
setreuid(*uid_t ruid*, *uid_t euid*);

DESCRIPTION

This interface is made obsolete by the saved ID functionality in setuid(2) and seteuid(2).

The real and effective user ID's of the current process are set according to the arguments. If the real user ID is changed, the saved user ID is changed to the new value of the effective user ID.

If *ruid* or *euid* is −1, the current uid is filled in by the system. Unprivileged users may change the real user ID to the effective user ID, and may change the effective user ID to the real user ID or the saved user ID; only the super-user may make other changes.

The **setreuid**() function has been used to swap the real and effective user IDs in set-user-ID programs to temporarily relinquish the set-user-ID value. This purpose is now better served by the use of the **seteuid**() function (see setuid(2)).

When setting the real and effective user IDs to the same value, this function is equivalent to the **setuid**() function. When setting only the effective user ID, this function is equivalent to the **seteuid**() function.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

[EPERM]	The current process is not the super-user and a change other than changing the effective user-id to the real user-id was specified.

SEE ALSO

getuid(2), seteuid(2), setgid(2), setuid(2)

HISTORY

The **setreuid**() function call appeared in 4.2BSD. An incompatible version was implemented in 4.4BSD. It was reimplemented in NetBSD 1.2 in a way compatible with 4.3BSD, SunOS and Linux, but should not be used in new code.

NAME

setsid — create session and set process group ID

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

pid_t
setsid(*void*);

DESCRIPTION

The **setsid** function creates a new session. The calling process is the session leader of the new session, is the process group leader of a new process group and has no controlling terminal. The calling process is the only process in either the session or the process group.

Upon successful completion, the **setsid** function returns the value of the process group ID of the new process group, which is the same as the process ID of the calling process.

ERRORS

If an error occurs, **setsid** returns −1 and the global variable *errno* is set to indicate the error, as follows:

[EPERM]	The calling process is already a process group leader, or the process group ID of a process other than the calling process matches the process ID of the calling process.

SEE ALSO

getsid(2), setpgid(2), tcgetpgrp(3), tcsetpgrp(3)

STANDARDS

The **setsid** function conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

NAME

setuid, **seteuid**, **setgid**, **setegid** — set user and group ID

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
setuid(*uid_t uid*);

int
seteuid(*uid_t euid*);

int
setgid(*gid_t gid*);

int
setegid(*gid_t egid*);

DESCRIPTION

The **setuid**() function sets the real and effective user IDs and the saved set-user-ID of the current process to the specified value. The **setuid**() function is permitted if the specified ID is equal to the real user ID of the process, or if the effective user ID is that of the super user.

The **setgid**() function sets the real and effective group IDs and the saved set-group-ID of the current process to the specified value. The **setgid**() function is permitted if the specified ID is equal to the real group ID of the process, or if the effective user ID is that of the super user. Supplementary group IDs remain unchanged.

The **seteuid**() function (**setegid**()) sets the effective user ID (group ID) of the current process. The effective user ID may be set to the value of the real user ID or the saved set-user-ID (see `intro`(2) and `execve`(2)); in this way, the effective user ID of a set-user-ID executable may be toggled by switching to the real user ID, then re-enabled by reverting to the set-user-ID value. Similarly, the effective group ID may be set to the value of the real group ID or the saved set-group-ID.

RETURN VALUES

Upon success, these functions return 0; otherwise −1 is returned.

If the user is not the super user, or the uid specified is not the real, effective ID, or saved ID, these functions return −1.

SEE ALSO

`getgid`(2), `getgroups`(2), `getuid`(2)

STANDARDS

The **setuid**() and **setgid**() functions are compliant with the ISO/IEC 9945-1:1990 (“POSIX.1”) specification with `_POSIX_SAVED_IDS` not defined. We do not implement the `_POSIX_SAVED_IDS` option as specified in the standard because this would make it impossible for a set-user-ID executable owned by a user other than the super-user to permanently revoke its privileges.

The **seteuid**() and **setegid**() functions are compliant with IEEE Std 1003.1-2001 (“POSIX.1”).

NAME

shmat, **shmdt** — map/unmap shared memory

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/shm.h>

*void **
shmat(*int shmid*, *const void *shmaddr*, *int shmflg*);

int
shmdt(*const void *shmaddr*);

DESCRIPTION

shmat() maps the shared memory segment associated with the shared memory identifier *shmid* into the address space of the calling process. The address at which the segment is mapped is determined by the *shmaddr* parameter. If it is equal to 0, the system will pick an address itself. Otherwise, an attempt is made to map the shared memory segment at the address *shmaddr* specifies. If SHM_RND is set in *shmflg*, the system will round the address down to a multiple of SHMLBA bytes (SHMLBA is defined in ⟨sys/shm.h⟩).

A shared memory segment can be mapped read-only by specifying the SHM_RDONLY flag in *shmflg*.

shmdt() unmaps the shared memory segment that is currently mapped at *shmaddr* from the calling process' address space. *shmaddr* must be a value returned by a prior **shmat**() call. A shared memory segment will remain in existence until it is removed by a call to shmctl(2) with the IPC_RMID command.

RETURN VALUES

shmat() returns the address at which the shared memory segment has been mapped into the calling process' address space when successful, **shmdt**() returns 0 on successful completion. Otherwise, a value of −1 is returned, and the global variable *errno* is set to indicate the error.

ERRORS

shmat() will fail if:

[EACCES]	The calling process has no permission to access this shared memory segment.
[ENOMEM]	There is not enough available data space for the calling process to map the shared memory segment.
[EINVAL]	*shmid* is not a valid shared memory identifier.
	shmaddr specifies an illegal address.
[EMFILE]	The number of shared memory segments has reached the system-wide limit.

shmdt() will fail if:

[EINVAL]	*shmaddr* is not the start address of a mapped shared memory segment.

SEE ALSO

ipcrm(1), ipcs(1), mmap(2), shmctl(2), shmget(2)

STANDARDS

The **shmat** and **shmdt** system calls conform to X/Open System Interfaces and Headers Issue 5 (“XSH5”).

HISTORY

Shared memory segments appeared in the first release of AT&T System V UNIX.

NAME

shmctl — shared memory control operations

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/shm.h>

int
shmctl(*int shmid*, *int cmd*, *struct shmid_ds *buf*);

DESCRIPTION

The **shmctl**() system call performs control operations on the shared memory segment specified by *shmid*.

Each shared memory segment has a **shmid_ds** structure associated with it which contains the following members:

```
struct ipc_perm shm_perm;   /* operation permissions */
size_t          shm_segsz;  /* size of segment in bytes */
pid_t           shm_lpid;   /* pid of last shm op */
pid_t           shm_cpid;   /* pid of creator */
shmatt_t        shm_nattch; /* # of current attaches */
time_t          shm_atime;  /* last shmat() time */
time_t          shm_dtime;  /* last shmdt() time */
time_t          shm_ctime;  /* last change by shmctl() */
```

The **ipc_perm** structure used inside the **shmid_ds** structure is defined in ⟨sys/ipc.h⟩ and contains the following members:

```
uid_t cuid;  /* creator user id */
gid_t cgid;  /* creator group id */
uid_t uid;  /* user id */
gid_t gid;  /* group id */
mode_t mode; /* permission (lower 9 bits) */
```

The operation to be performed by **shmctl**() is specified in *cmd* and is one of:

IPC_STAT   Gather information about the shared memory segment and place it in the structure pointed to by *buf*.

IPC_SET   Set the value of the *shm_perm.uid*, *shm_perm.gid* and *shm_perm.mode* fields in the structure associated with *shmid*. The values are taken from the corresponding fields in the structure pointed to by *buf*. This operation can only be executed by the super-user, or a process that has an effective user id equal to either *shm_perm.cuid* or *shm_perm.uid* in the data structure associated with the shared memory segment.

IPC_RMID   Remove the shared memory segment specified by *shmid* and destroy the data associated with it. Only the super-user or a process with an effective uid equal to the *shm_perm.cuid* or *shm_perm.uid* values in the data structure associated with the queue can do this.

SHM_LOCK   Lock the shared memory segment specified by *shmid* in memory. This operation can only be executed by the super-user.

SHM_UNLOCK
Unlock the shared memory segment specified by *shmid*. This operation can only be executed by the super-user.

The read and write permissions on a shared memory identifier are determined by the *shm_perm.mode* field in the same way as is done with files (see `chmod`(2)), but the effective uid can match either the *shm_perm.cuid* field or the *shm_perm.uid* field, and the effective gid can match either *shm_perm.cgid* or *shm_perm.gid*.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

shmctl() will fail if:

[EACCES]	The command is `IPC_STAT` and the caller has no read permission for this shared memory segment.
[EFAULT]	*buf* specifies an invalid address.
[EINVAL]	*shmid* is not a valid shared memory segment identifier.
	cmd is not a valid command.
[ENOMEM]	The *cmd* is equal to `SHM_LOCK` and there is not enough physical memory.
[EPERM]	*cmd* is equal to `IPC_SET` or `IPC_RMID` and the caller is not the super-user, nor does the effective uid match either the *shm_perm.uid* or *shm_perm.cuid* fields of the data structure associated with the shared memory segment.
	An attempt was made to increase the value of *shm_qbytes* through `IPC_SET` but the caller is not the super-user.
	The *cmd* is equal to `SHM_LOCK` or `SHM_UNLOCK` and the caller is not the super-user.

SEE ALSO

`ipcrm`(1), `ipcs`(1), `shmat`(2), `shmget`(2)

STANDARDS

The **shmctl** system call conforms to X/Open System Interfaces and Headers Issue 5 (“XSH5”).

HISTORY

Shared memory segments appeared in the first release of AT&T System V UNIX.

NAME

shmget — get shared memory segment

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/shm.h>

int
shmget(*key_t key*, *size_t size*, *int shmflg*);

DESCRIPTION

shmget() returns the shared memory identifier associated with the key *key*.

A shared memory segment is created if either *key* is equal to IPC_PRIVATE, or *key* does not have a shared memory segment identifier associated with it, and the IPC_CREAT bit is set in *shmflg*. If both the IPC_CREAT bit and the IPC_EXCL bit are set in *shmflg*, and *key* has a shared memory segment identifier associated with it already, the operation will fail.

If a new shared memory segment is created, the data structure associated with it (the *shmid_ds* structure, see shmctl(2)) is initialized as follows:

- *shm_perm.cuid* and *shm_perm.uid* are set to the effective uid of the calling process.
- *shm_perm.gid* and *shm_perm.cgid* are set to the effective gid of the calling process.
- *shm_perm.mode* is set to the lower 9 bits of *shmflg*.
- *shm_lpid*, *shm_nattch*, *shm_atime*, and *shm_dtime* are set to 0.
- *shm_ctime* is set to the current time.
- *shm_segsz* is set to the value of *size*.

RETURN VALUES

Upon successful completion a positive shared memory segment identifier is returned. Otherwise, −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EACCES]	A shared memory segment is already associated with *key* and the caller has no permission to access it.
[EEXIST]	Both IPC_CREAT and IPC_EXCL are set in *shmflg*, and a shared memory segment is already associated with *key*.
[ENOSPC]	A new shared memory identifier could not be created because the system limit for the number of shared memory identifiers has been reached.
[ENOENT]	IPC_CREAT is not set in *shmflg* and no shared memory segment associated with *key* was found.
[ENOMEM]	There is not enough memory left to create a shared memory segment of the requested size.

SEE ALSO

ipcrm(1), ipcs(1), mmap(2), shmat(2), shmctl(2), ftok(3)

STANDARDS

The **shmget** system call conforms to X/Open System Interfaces and Headers Issue 5 (“XSH5”).

HISTORY

Shared memory segments appeared in the first release of AT&T System V UNIX.

NAME

shutdown — shut down part of a full-duplex connection

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/socket.h>

int
shutdown(*int s* , *int how*);

DESCRIPTION

The **shutdown**() call causes all or part of a full-duplex connection on the socket associated with *s* to be shut down. The *how* argument specifies which part of the connection will be shut down. Permissible values are:

SHUT_RD	further receives will be disallowed.
SHUT_WR	further sends will be disallowed.
SHUT_RDWR	further sends and receives will be disallowed.

RETURN VALUES

A 0 is returned if the call succeeds, −1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	*s* is not a valid descriptor.
[EINVAL]	The *how* argument is invalid.
[ENOTCONN]	The specified socket is not connected.
[ENOTSOCK]	*s* is a file, not a socket.

SEE ALSO

connect(2), socket(2)

HISTORY

The **shutdown**() function call appeared in 4.2BSD. The *how* arguments used to be simply 0, 1, and 2, but now have named values as specified by X/Open Portability Guide Issue 4 (“XPG4”).

NAME

sigaction — software signal facilities

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <signal.h>

int
sigaction(int sig, const struct sigaction * restrict act,
    struct sigaction * restrict oact);
```

DESCRIPTION

The system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify a *handler* to which a signal is delivered, or specify that a signal is to be *ignored*. A process may also specify that a default action is to be taken by the system when a signal occurs. A signal may also be *blocked*, in which case its delivery is postponed until it is *unblocked*. The action to be taken on delivery is determined at the time of delivery. Normally, signal handlers execute on the current stack of the process. This may be changed, on a per-handler basis, so that signals are taken on a special *signal stack*.

Signal routines execute with the signal that caused their invocation *blocked*, but other signals may yet occur. A global *signal mask* defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally empty). It may be changed with a `sigprocmask`(2) call, or when a signal is delivered to the process. Signal masks are represented using the *sigset_t* type; the `sigsetops`(3) interface is used to modify such data.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently *blocked* by the process then it is delivered to the process. Signals may be delivered any time a process enters the operating system (e.g., during a system call, page fault or trap, or clock interrupt). If multiple signals are ready to be delivered at the same time, any signals that could be caused by traps are delivered first. Additional signals may be processed at the same time, with each appearing to interrupt the handlers for the previous signals before their first instructions. The set of pending signals is returned by the `sigpending`(2) function. When a caught signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described below), and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally the process will resume execution in the context from before the signal's delivery. If the process wishes to resume in a different context, then it must arrange to restore the previous context itself.

struct sigaction includes the following members:

```
void      (*sa_sigaction)(int sig, siginfo_t *info, void *ctx);
void      (*sa_handler)(int sig);
sigset_t  sa_mask;
int       sa_flags;
```

When a signal is delivered to a process a new signal mask is installed for the duration of the process' signal handler (or until a `sigprocmask`(2) call is made). This mask is formed by taking the union of the current signal mask, the signal to be delivered, and the signal mask associated with the handler to be invoked, *sa_mask*.

sigaction() assigns an action for a specific signal. If *act* is non-zero, it specifies an action (`SIG_DFL`, `SIG_IGN`, or a handler routine) and mask to be used when delivering the specified signal. If *oact* is non-zero, the previous handling information for the signal is returned to the user.

Once a signal handler is installed, it remains installed until another **sigaction**() call is made, or an `execve`(2) is performed. A signal-specific default action may be reset by setting *sa_handler* to `SIG_DFL`. The defaults are process termination, possibly with core dump; no action; stopping the process; or continuing the process. See the signal list below for each signal's default action. If *sa_handler* is set to `SIG_DFL`, the default action for the signal is to discard the signal, and if a signal is pending, the pending signal is discarded even if the signal is masked. If *sa_handler* is set to `SIG_IGN`, current and pending instances of the signal are ignored and discarded.

Options may be specified by setting *sa_flags*.

`SA_NODEFER` If set, then the signal that caused the handler to be executed is not added to the list of block signals. Please note that *sa_mask* takes precedence over `SA_NODEFER`, so that if the specified signal is blocked in *sa_mask*, then `SA_NODEFER` will have no effect.

`SA_NOCLDSTOP` If set when installing a catching function for the `SIGCHLD` signal, the `SIGCHLD` signal will be generated only when a child process exits, not when a child process stops.

`SA_NOCLDWAIT` If set, the system will not create a zombie when the child exits, but the child process will be automatically waited for. The same effect can be achieved by setting the signal handler for `SIGCHLD` to `SIG_IGN`.

`SA_ONSTACK` If set, the system will deliver the signal to the process on a *signal stack*, specified with `sigaltstack`(2).

`SA_RESETHAND` If set, the default action will be reinstated when the signal is first posted.

`SA_RESTART` Normally, if a signal is caught during the system calls listed below, the call may be forced to terminate with the error `EINTR`, the call may return with a data transfer shorter than requested, or the call may be restarted. Restarting of pending calls is requested by setting the `SA_RESTART` bit in *sa_flags*. The affected system calls include `open`(2), `read`(2), `write`(2), `sendto`(2), `recvfrom`(2), `sendmsg`(2) and `recvmsg`(2) on a communications channel or a slow device (such as a terminal, but not a regular file) and during a `wait`(2) or `ioctl`(2). However, calls that have already committed are not restarted, but instead return a partial success (for example, a short read count).

After a `fork`(2) or `vfork`(2) all signals, the signal mask, the signal stack, and the restart/interrupt flags are inherited by the child.

The `execve`(2) system call reinstates the default action for all signals which were caught and resets all signals to be caught on the user stack. Ignored signals remain ignored; the signal mask remains the same; signals that restart pending system calls continue to do so.

See `signal`(7) for comprehensive list of supported signals.

`SA_SIGINFO` If set, the signal handler function will receive additional information about the caught signal. An alternative handler that gets passed additional arguments will be called which is named *sa_sigaction*. The *sig* argument of this handler contains the signal number that was caught. The *info* argument contains additional signal specific information which is listed in `siginfo`(2). The *ctx* argument is a pointer to the `ucontext`(2) context where the signal handler will return to.

`SA_NOKERNINFO` This flag is relevant only to `SIGINFO`, and turns off printing kernel messages on the tty. It is similar to the `NOKERNINFO` flag in `termios`(4).

Only functions that are async-signal-safe can safely be used in signal handlers, see `signal`(7) for a complete list.

NOTES

The mask specified in *`act`* is not allowed to block `SIGKILL` or `SIGSTOP`. This is enforced silently by the system.

RETURN VALUES

A 0 value indicates that the call succeeded. A −1 return value indicates an error occurred and *errno* is set to indicate the reason.

ERRORS

`sigaction`() will fail and no new signal handler will be installed if one of the following occurs:

[EFAULT]	Either *`act`* or *`oact`* points to memory that is not a valid part of the process address space.
[EINVAL]	*`sig`* is not a valid signal number.
[EINVAL]	An attempt is made to ignore or supply a handler for `SIGKILL` or `SIGSTOP`.
[EINVAL]	The *sa_flags* word contains bits other than `SA_NOCLDSTOP`, `SA_NOCLDWAIT`, `SA_NODEFER`, `SA_ONSTACK`, `SA_RESETHAND`, `SA_RESTART`, and `SA_SIGINFO`.

SEE ALSO

`kill`(1), `kill`(2), `ptrace`(2), `sigaltstack`(2), `siginfo`(2), `sigprocmask`(2), `sigsuspend`(2), `setjmp`(3), `sigsetops`(3), `tty`(4), `signal`(7)

STANDARDS

The **`sigaction`**() function conforms to ISO/IEC 9945-1:1990 (“POSIX.1”). The `SA_ONSTACK` and `SA_RESTART` flags are Berkeley extensions, available on most BSD−derived systems.

NAME

sigaltstack — set and/or get signal stack context

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <signal.h>

```
typedef struct sigaltstack {
       void   *ss_sp;
       size_t  ss_size;
       int     ss_flags;
} stack_t;
```

int
sigaltstack(*const stack_t * restrict ss*, *stack_t * restrict oss*);

DESCRIPTION

sigaltstack() allows users to define an alternative stack on which signals are to be processed. If *ss* is non-zero, it specifies a pointer to and the size of a *signal stack* on which to deliver signals, and tells the system if the process is currently executing on that stack. When a signal's action indicates its handler should execute on the signal stack (specified with a sigaction(2) call), the system checks to see if the process is currently executing on that stack. If the process is not currently executing on the signal stack, the system arranges a switch to the signal stack for the duration of the signal handler's execution.

If SS_DISABLE is set in *ss_flags*, *ss_sp* and *ss_size* are ignored and the signal stack will be disabled. Trying to disable an active stack will cause **sigaltstack** to return −1 with *errno* set to EINVAL. A disabled stack will cause all signals to be taken on the regular user stack. If the stack is later re-enabled then all signals that were specified to be processed on an alternative stack will resume doing so.

If *oss* is non-zero, the current signal stack state is returned. The *ss_flags* field will contain the value SS_ONSTACK if the process is currently on a signal stack and SS_DISABLE if the signal stack is currently disabled.

NOTES

The value SIGSTKSZ is defined to be the number of bytes/chars that would be used to cover the usual case when allocating an alternative stack area. The following code fragment is typically used to allocate an alternative stack.

```
if ((sigstk.ss_sp = malloc(SIGSTKSZ)) == NULL)
        /* error return */
sigstk.ss_size = SIGSTKSZ;
sigstk.ss_flags = 0;
if (sigaltstack(&sigstk,0) < 0)
        perror("sigaltstack");
```

An alternative approach is provided for programs with signal handlers that require a specific amount of stack space other than the default size. The value MINSIGSTKSZ is defined to be the number of bytes/chars that is required by the operating system to implement the alternative stack feature. In computing an alternative stack size, programs should add MINSIGSTKSZ to their stack requirements to allow for the operating system overhead.

Signal stacks are automatically adjusted for the direction of stack growth and alignment requirements. Signal stacks may or may not be protected by the hardware and are not ‘‘grown’’ automatically as is done for the normal stack. If the stack overflows and this space is not protected unpredictable results may occur.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

sigaltstack() will fail and the signal stack context will remain unchanged if one of the following occurs.

`[EFAULT]`	Either *ss* or *oss* points to memory that is not a valid part of the process address space.
`[EINVAL]`	An attempt was made to disable an active stack.
`[ENOMEM]`	Size of alternative stack area is less than `MINSIGSTKSZ`.

SEE ALSO

`sigaction`(2), `setjmp`(3), `signal`(7)

STANDARDS

The **sigaltstack**() function conforms to X/Open Portability Guide Issue 4, Version 2 (“XPG4.2”).

HISTORY

The predecessor to **sigaltstack**, the **sigstack**() system call, appeared in 4.2BSD.

NAME

siginfo — signal information

SYNOPSIS

#include <signal.h>

DESCRIPTION

siginfo is a structure type which contains information about a signal delivered to a process.

siginfo includes the following members:

```
int si_signo;
int si_errno;
int si_code;
```

si_signo contains the signal number generated by the system.

If *si_errno* is non-zero, then it contains a system specific error number associated with this signal. This number is defined in errno(2).

If *si_code* is less than or equal to zero, the signal was generated by a user process or a user requested service:

SI_USER The signal was generated via kill(2). The **siginfo** structure contains the following additional members:

```
pid_t si_pid;
uid_t si_uid;
```

The *si_pid* field contains the pid of the sending process and the *si_uid* field contains the user id of the sending process.

SI_TIMER The signal was generated because a timer set by timer_settime(2) has expired. The **siginfo** structure contains the following additional members:

```
sigval_t si_value;
```

The *si_value* field contains the value set via timer_create(2).

SI_ASYNCIO The signal was generated by completion of an asynchronous I/O operation. The **siginfo** structure contains the following additional members:

```
int si_fd;
long si_band;
```

The *si_fd* argument contains the file descriptor number on which the operation was completed and the *si_band* field contains the side and priority of the operation. If the operation was a normal read, *si_band* will contain POLLIN | POLLRDNORM; on an out-of-band read it will contain POLLPRI | POLLRDBAND; on a normal write it will contain POLLOUT | POLLWRNORM; on an out-of-band write it will contain POLLPRI | POLLWRBAND.

If *si_code* is positive, then it contains a signal specific reason why the signal was generated:

SIGILL

ILL_ILLOPC	Illegal opcode
ILL_ILLOPN	Illegal operand

ILL_ILLADR	Illegal addressing mode
ILL_ILLTRP	Illegal trap
ILL_PRVOPC	Privileged opcode
ILL_PRVREG	Privileged register
ILL_COPROC	Coprocessor error
ILL_BADSTK	Internal stack error

SIGFPE

FPE_INTDIV	Integer divide by zero
FPE_INTOVF	Integer overflow
FPE_FLTDIV	Floating point divide by zero
FPE_FLTOVF	Floating point overflow
FPE_FLTUND	Floating point underflow
FPE_FLTRES	Floating poing inexact result
FPE_FLTINV	Invalid Floating poing operation
FPE_FLTSUB	Subscript out of range

SIGSEGV

SEGV_MAPERR	Address not mapped to object
SEGV_ACCERR	Invalid permissions for mapped object

SIGBUS

BUS_ADRALN	Invalid address alignment
BUS_ADRERR	Non-existant physical address
BUS_OBJERR	Object specific hardware error

SIGTRAP

TRAP_BRKPT	Process breakpoint
TRAP_TRACE	Process trace trap

SIGCHLD

CLD_EXITED	Child has exited

CLD_KILLED	Child has terminated abnormally but did not create a core file
CLD_DUMPED	Child has terminated abnormally and created a core file
CLD_TRAPPED	Traced child has trapped
CLD_STOPPED	Child has stopped
CLD_CONTINUED	Stopped child has continued

SIGPOLL

POLL_IN	Data input available
POLL_OUT	Output buffers available
POLL_MSG	Input message available
POLL_ERR	I/O Error
POLL_PRI	High priority input available
POLL_HUP	Device disconnected

For `SIGILL`, `SIGFPE`, and `SIGTRAP` the **`siginfo`** structure contains the following additional members:

```
void *si_addr;
int si_trap;
```

`si_addr` contains the address of the faulting instruction and *`si_trap`* contains a hardware specific reason.

For `SIGBUS` and `SIGSEGV` the **`siginfo`** structure contains the following additional members:

```
void *si_addr;
int si_trap;
```

`si_addr` contains the address of the faulting data and *`si_trap`* contains a hardware specific reason.

For `SIGPOLL` the **`siginfo`** structure contains the following additional members:

```
int si_fd;
long si_band;
```

The *`si_fd`* argument contains the file descriptor number on which the operation was completed and the *`si_band`* field contains the side and priority of the operation as described above.

Finally, for `SIGCHLD` the **`siginfo`** structure contains the following additional members:

```
pid_t si_pid;
uid_t si_uid;
int si_status;
clock_t si_utime;
clock_t si_stime;
```

The *`si_pid`* field contains the pid of the process who's status changed, the *`si_uid`* field contains the user id of the that process, the *`si_status`* field contains a status code described in `waitpid`(2), and the *`si_utime`* and *`si_stime`* fields contain the user and system process accounting time.

STANDARDS

The **`siginfo`** type conforms to X/Open System Interfaces and Headers Issue 5 ("XSH 5").

HISTORY

The **`siginfo`** functionality first appeared in AT&T System V.4 UNIX.

NAME

sigpending — get pending signals

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <signal.h>

int
sigpending(*sigset_t *set*);

DESCRIPTION

The **sigpending** function returns a mask of the signals pending for delivery to the calling process in the location indicated by *set*. Signals may be pending because they are currently masked, or they are in transition before delivery (although the latter case is not normally detectable).

RETURN VALUES

A 0 value indicates that the call succeeded. A −1 return value indicates an error occurred and *errno* is set to indicate the reason.

ERRORS

The **sigpending** function does not currently detect any errors.

SEE ALSO

sigaction(2), sigprocmask(2), signal(7)

STANDARDS

The **sigpending** function conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

NAME

sigprocmask — manipulate current signal mask

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <signal.h>

int
sigprocmask(*int how*, *const sigset_t * restrict set*,
    *sigset_t * restrict oset*);

DESCRIPTION

The **sigprocmask**() function examines and/or changes the current signal mask (those signals that are blocked from delivery). Signals are blocked if they are members of the current signal mask set.

If *set* is not null, the action of **sigprocmask**() depends on the value of the parameter *how*. The signal mask is changed as a function of the specified *set* and the current mask. The function is specified by *how* using one of the following values:

SIG_BLOCK	The new mask is the union of the current mask and the specified *set*.
SIG_UNBLOCK	The new mask is the intersection of the current mask and the complement of the specified *set*.
SIG_SETMASK	The current mask is replaced by the specified *set*.

If *oset* is not null, it is set to the previous value of the signal mask.

When *set* is null, the value of *how* is insignificant and the mask remains unset providing a way to examine the signal mask without modification.

The system quietly disallows SIGKILL or SIGSTOP to be blocked.

RETURN VALUES

A 0 value indicates that the call succeeded. A −1 return value indicates an error occurred and *errno* is set to indicate the reason.

ERRORS

The **sigprocmask**() call will fail and the signal mask will be unchanged if one of the following occurs:

[EINVAL]	*how* has a value other than those listed here.

SEE ALSO

kill(2), sigaction(2), sigsuspend(2), pthread_sigmask(3), sigsetops(3), signal(7)

STANDARDS

The **sigprocmask**() function conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

NAME

sigstack — set and/or get signal stack context

SYNOPSIS

#include <signal.h>

int
sigstack(*const struct sigstack **, *struct sigstack **);

DESCRIPTION

The **sigstack**() function has been deprecated in favor of the interface described in sigaltstack(2).

SEE ALSO

sigaltstack(2), signal(7)

HISTORY

The **sigstack** function call appeared in 4.2BSD.

NAME

sigsuspend — atomically release blocked signals and wait for interrupt

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <signal.h>

int
sigsuspend(*const sigset_t *sigmask*);

DESCRIPTION

sigsuspend() temporarily changes the blocked signal mask to the set to which *sigmask* points, and then waits for a signal to arrive; on return the previous set of masked signals is restored. The signal mask set is usually empty to indicate that all signals are to be unblocked for the duration of the call.

In normal usage, a signal is blocked using sigprocmask(2) to begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses awaiting work by using **sigsuspend**() with the previous mask returned by sigprocmask(2).

RETURN VALUES

The **sigsuspend**() function always terminates by being interrupted, returning −1 with *errno* set to EINTR.

SEE ALSO

sigaction(2), sigprocmask(2), sigsetops(3), signal(7)

STANDARDS

The **sigsuspend** function call conforms to ISO/IEC 9945-1:1990 (“POSIX.1”).

NAME

sigtimedwait, **sigwaitinfo**, **sigwait** — wait for queued signals

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <signal.h>

int
sigtimedwait(*const sigset_t * restrict set*, *siginfo_t * restrict info*,
    *const struct timespec * restrict timeout*);

int
sigwaitinfo(*const sigset_t * restrict set*, *siginfo_t * restrict info*);

int
sigwait(*const sigset_t * restrict set*, *int * restrict sig*);

DESCRIPTION

sigwaitinfo() and **sigwait**() return the first pending signal from the set specified by *set*. Should multiple signals from *set* be pending, the lowest numbered one is returned. The selection order between realtime and non-realtime signals is unspecified. If there is no signal from *set* pending at the time of the call, the calling thread is suspended until one of the specified signals is generated.

sigtimedwait() is exactly equal to **sigwaitinfo**(), except *timeout* specifies the maximum time interval for which the calling thread will be suspended. If *timeout* is zero (tv_sec == tv_nsec == 0), **sigtimedwait**() only checks the currently pending signals and returns immediately. If NULL is used for *timeout*, **sigtimedwait**() behaves exactly like **sigwaitinfo**() in all regards.

If several threads are waiting for a given signal, exactly one of them returns from the signal wait when the signal is generated.

Behaviour of these functions is unspecified if any of the signals in *set* are unblocked at the time these functions are called.

RETURN VALUES

Upon successful completion *info* is updated with signal information, and the function returns 0. Otherwise, −1 is returned and the global variable *errno* indicates the error.

ERRORS

sigwaitinfo() and **sigwait**() always succeed.

sigtimedwait() will fail and the *info* pointer will remain unchanged if:

[EAGAIN]	No signal specified in *set* was generated in the specified *timeout*.

sigtimedwait() may also fail if:

[EINVAL]	The specified *timeout* was invalid.

This error is only checked if no signal from *set* is pending and it would be necessary to wait.

SEE ALSO

sigaction(2), sigprocmask(2), signal(7)

STANDARDS

The functions **sigtimedwait**(), **sigwaitinfo**(), and **sigwait**() conform to IEEE Std 1003.1-2001 (“POSIX.1”).

HISTORY

The **sigtimedwait**(), **sigwaitinfo**(), and **sigwait**() functions appeared in NetBSD 2.0.

AUTHORS

The initial NetBSD implementation of the signal wait functions was written by Jaromir Dolecek ⟨jdolecek@NetBSD.org⟩.

NAME

socket — create an endpoint for communication

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/socket.h>

int
socket(*int domain*, *int type*, *int protocol*);

DESCRIPTION

socket() creates an endpoint for communication and returns a descriptor.

The *domain* parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. These families are defined in the include file ⟨sys/socket.h⟩. The currently understood formats are:

PF_LOCAL	local (previously UNIX) domain protocols
PF_INET	ARPA Internet protocols
PF_INET6	IPv6 (Internet Protocol version 6) protocols
PF_ISO	ISO protocols
PF_NS	Xerox Network Systems protocols
PF_IMPLINK	IMP host at IMP link layer
PF_APPLETALK	AppleTalk protocols
PF_BLUETOOTH	Bluetooth protocols

The socket has the indicated *type*, which specifies the semantics of communication. Currently defined types are:

```
SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET
SOCK_RDM
```

A SOCK_STREAM type provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism may be supported. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). A SOCK_SEQPACKET socket may provide a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer may be required to read an entire packet with each read system call. This facility is protocol specific, and presently implemented only for PF_NS. SOCK_RAW sockets provide access to internal network protocols and interfaces. The types SOCK_RAW, which is available only to the super-user, and SOCK_RDM, which is planned, but not yet implemented, are not described here.

The *protocol* specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the communication domain in which communication is to take place; see protocols(5).

Sockets of type SOCK_STREAM are full-duplex byte streams. A stream socket must be in a *connected* state before any data may be sent or received on it. A connection to another socket is created with a connect(2) call. Once connected, data may be transferred using read(2) and write(2) calls or some variant of the send(2) and recv(2) calls. When a session has been completed a close(2) may be performed. Out-of-band data may also be transmitted as described in `send`(2) and received as described in `recv`(2).

The communications protocols used to implement a `SOCK_STREAM` ensure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with −1 returns and with `ETIMEDOUT` as the specific code in the global variable *errno*. The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for an extended period (e.g., 5 minutes). A `SIGPIPE` signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

`SOCK_SEQPACKET` sockets employ the same system calls as `SOCK_STREAM` sockets. The only difference is that `read`(2) calls will return only the amount of data requested, and any remaining in the arriving packet will be discarded.

`SOCK_DGRAM` and `SOCK_RAW` sockets allow sending of datagrams to correspondents named in `send`(2) calls. Datagrams are generally received with `recvfrom`(2), which returns the next datagram with its return address.

An `fcntl`(2) call can be used to specify a process group to receive a `SIGURG` signal when the out-of-band data arrives. It may also enable non-blocking I/O and asynchronous notification of I/O events via `SIGIO`.

The operation of sockets is controlled by socket level *options*. These options are defined in the file ⟨`sys/socket.h`⟩. The `setsockopt`(2) and `getsockopt`(2) system calls are used to set and get options, respectively.

RETURN VALUES

A −1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

ERRORS

The **`socket`**() call fails if:

`[EACCES]`	Permission to create a socket of the specified type and/or protocol is denied.
`[EAFNOSUPPORT]`	The address family (domain) is not supported or the specified domain is not supported by this protocol family.
`[EMFILE]`	The per-process descriptor table is full.
`[ENFILE]`	The system file table is full.
`[ENOBUFS]`	Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.
`[EPROTONOSUPPORT]`	The protocol family is not supported or the specified protocol is not supported within this domain.
`[EPROTOTYPE]`	The socket type is not supported by the protocol.

SEE ALSO

`accept`(2), `bind`(2), `connect`(2), `getsockname`(2), `getsockopt`(2), `ioctl`(2), `listen`(2), `poll`(2), `read`(2), `recv`(2), `select`(2), `send`(2), `setsockopt`(2), `shutdown`(2), `socketpair`(2), `write`(2), `getprotoent`(3)

Stuart Sechrest, *An Introductory 4.4BSD Interprocess Communication Tutorial*. (see `/usr/share/doc/psd/20.ipctut`)

Samuel J. Leffler, Robert S. Fabry, William N. Joy, Phil Lapsley, Steve Miller, and Chris Torek, *Advanced 4.4BSD IPC Tutorial*. (see `/usr/share/doc/psd/21.ipc`)

HISTORY

The **`socket`**() function call appeared in 4.2BSD.

NAME

socketpair — create a pair of connected sockets

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/socket.h>

int
socketpair(*int d*, *int type*, *int protocol*, *int *sv*);

DESCRIPTION

The **socketpair**() call creates an unnamed pair of connected sockets in the specified domain *d*, of the specified *type*, and using the optionally specified *protocol*. The descriptors used in referencing the new sockets are returned in *sv*[0] and *sv*[1]. The two sockets are indistinguishable.

RETURN VALUES

A 0 is returned if the call succeeds, −1 if it fails.

ERRORS

The call succeeds unless:

[EMFILE]	Too many descriptors are in use by this process.
[ENFILE]	The system file table is full.
[EAFNOSUPPORT]	The specified address family is not supported on this machine.
[EPROTONOSUPPORT]	The specified protocol is not supported on this machine.
[EOPNOTSUPP]	The specified protocol does not support creation of socket pairs.
[EFAULT]	The address *sv* does not specify a valid part of the process address space.

SEE ALSO

pipe(2), read(2), write(2)

HISTORY

The **socketpair**() function call appeared in 4.2BSD.

BUGS

This call is currently implemented only for the LOCAL domain. Many operating systems only accept a *protocol* of PF_UNSPEC, so that should be used instead of PF_LOCAL for maximal portability.

NAME

stat, **lstat**, **fstat** — get file status

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/stat.h>

int
stat(*const char *path*, *struct stat *sb*);

int
lstat(*const char *path*, *struct stat *sb*);

int
fstat(*int fd*, *struct stat *sb*);

DESCRIPTION

The **stat**() function obtains information about the file pointed to by *path*. Read, write or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable.

lstat() is like **stat**() except in the case where the named file is a symbolic link, in which case **lstat**() returns information about the link, while **stat**() returns information about the file the link references.

The **fstat**() function obtains the same information about an open file known by the file descriptor *fd*.

The *sb* argument is a pointer to a *stat* structure as defined by ⟨sys/stat.h⟩ (shown below) and into which information is placed concerning the file.

```
struct stat {
    dev_t     st_dev;     /* device containing the file */
    ino_t     st_ino;     /* file's serial number */
    mode_t    st_mode;    /* file's mode (protection and type) */
    nlink_t   st_nlink;   /* number of hard links to the file */
    uid_t     st_uid;     /* user-id of owner */
    gid_t     st_gid;     /* group-id of owner */
    dev_t     st_rdev;    /* device type, for device special file */
#if defined(_NETBSD_SOURCE)
    struct timespec st_atimespec;  /* time of last access */
    struct timespec st_mtimespec;  /* time of last data modification */
    struct timespec st_ctimespec;  /* time of last file status change */
#else
    time_t    st_atime;            /* time of last access */
    long      st_atimensec;        /* nsec of last access */
    time_t    st_mtime;            /* time of last data modification */
    long      st_mtimensec;        /* nsec of last data modification */
    time_t    st_ctime;            /* time of last file status change */
    long      st_ctimensec;        /* nsec of last file status change */
#endif
    off_t     st_size;    /* file size, in bytes */
    blkcnt_t  st_blocks;  /* blocks allocated for file */
    blksize_t st_blksize; /* optimal file sys I/O ops blocksize */
    uint32_t st_flags;   /* user defined flags for file */
    uint32_t st_gen;     /* file generation number */
#if defined(_NETBSD_SOURCE)
    struct timespec st_birthtimespec; /* time of inode creation */
#else
    time_t    st_birthtime;             /* time of inode creation */
    long      st_birthtimensec;         /* nsec of inode creation */
#endif
};
```

The time-related fields of *struct stat* are as follows:

st_atime	Time when file data was last accessed. Changed by the `mknod`(2), `utimes`(2) and `read`(2) system calls.
st_mtime	Time when file data was last modified. Changed by the `mknod`(2), `utimes`(2) and `write`(2) system calls.
st_ctime	Time when file status was last changed (file metadata modification). Changed by the `chflags`(2), `chmod`(2), `chown`(2), `link`(2), `mknod`(2), `rename`(2), `unlink`(2), `utimes`(2) and `write`(2) system calls.
st_birthtime	Time when the inode was created.

If `_NETBSD_SOURCE` is defined, the time-related fields are defined as:

```
#if defined(_NETBSD_SOURCE)
#define st_atime                st_atimespec.tv_sec
#define st_atimensec            st_atimespec.tv_nsec
#define st_mtime                st_mtimespec.tv_sec
#define st_mtimensec            st_mtimespec.tv_nsec
#define st_ctime                st_ctimespec.tv_sec
#define st_ctimensec            st_ctimespec.tv_nsec
#define st_birthtime            st_birthtimespec.tv_sec
#define st_birthtimensec        st_birthtimespec.tv_nsec
#endif
```

The size-related fields of the *struct stat* are as follows:

st_size	The size of the file in bytes. A directory will be a multiple of the size of the `dirent`(5) structure. Some filesystems (notably ZFS) return the number of enties in the directory instead of the size in bytes.
st_blksize	The optimal I/O block size for the file.
st_blocks	The actual number of blocks allocated for the file in 512-byte units. As short symbolic links are stored in the inode, this number may be zero.

The status information word *st_mode* has the following bits:

```
#define S_IFMT 0170000           /* type of file */
#define        S_IFIFO  0010000  /* named pipe (fifo) */
#define        S_IFCHR  0020000  /* character special */
#define        S_IFDIR  0040000  /* directory */
#define        S_IFBLK  0060000  /* block special */
#define        S_IFREG  0100000  /* regular */
#define        S_IFLNK  0120000  /* symbolic link */
#define        S_IFSOCK 0140000  /* socket */
#define        S_IFWHT  0160000  /* whiteout */
```

```
#define S_ISUID 0004000  /* set user id on execution */
#define S_ISGID 0002000  /* set group id on execution */
#define S_ISVTX 0001000  /* save swapped text even after use */
#define S_IRUSR 0000400  /* read permission, owner */
#define S_IWUSR 0000200  /* write permission, owner */
#define S_IXUSR 0000100  /* execute/search permission, owner */
#define S_IRGRP 0000040  /* read permission, group */
#define S_IWGRP 0000020  /* write permission, group */
#define S_IXGRP 0000010  /* execute/search permission, group */
#define S_IROTH 0000004  /* read permission, other */
#define S_IWOTH 0000002  /* write permission, other */
#define S_IXOTH 0000001  /* execute/search permission, other */
```

For a list of access modes, see ⟨sys/stat.h⟩, access(2) and chmod(2).

The status information word *st_flags* has the following bits:

```
#define UF_NODUMP      0x00000001 /* do not dump file */
#define UF_IMMUTABLE   0x00000002 /* file may not be changed */
#define UF_APPEND      0x00000004 /* writes to file may only append */
#define UF_OPAQUE      0x00000008 /* directory is opaque wrt. union */
#define SF_ARCHIVED    0x00010000 /* file is archived */
#define SF_IMMUTABLE   0x00020000 /* file may not be changed */
#define SF_APPEND      0x00040000 /* writes to file may only append */
```

For a description of the flags, see chflags(2).

RETURN VALUES

Upon successful completion a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

COMPATIBILITY

Previous versions of the system used different types for the st_dev, st_uid, st_gid, st_rdev, st_size, st_blksize and st_blocks fields.

ERRORS

stat() and **lstat**() will fail if:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EFAULT]	*sb* or *name* points to an invalid address.
[ENXIO]	The named file is a character special or block special file, and the device associated with this special file does not exist.
[EIO]	An I/O error occurred while reading from or writing to the file system.

`[EBADF]`	A badly formed v-node was encountered. This can happen if a file system information node is incorrect.

`fstat`() will fail if:

`[EBADF]`	*`fd`* is not a valid open file descriptor.
`[EFAULT]`	*`sb`* points to an invalid address.
`[EIO]`	An I/O error occurred while reading from or writing to the file system.

SEE ALSO

`chflags`(2), `chmod`(2), `chown`(2), `utimes`(2), `dir`(5), `symlink`(7)

STANDARDS

The **`stat`**() and **`fstat`**() functions conform to ISO/IEC 9945-1:1990 (“POSIX.1”).

HISTORY

A **`lstat`**() function call appeared in 4.2BSD.

BUGS

Applying **`fstat`**() to a socket (and thus to a pipe) returns a zero’d buffer, except for the blocksize field, and a unique device and file serial number.

NAME

statvfs, **statvfs1**, **fstatvfs**, **fstatvfs1** — get file system statistics

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/statvfs.h>

int
statvfs(*const char *path*, *struct statvfs *buf*);

int
statvfs1(*const char *path*, *struct statvfs *buf*, *int flags*);

int
fstatvfs(*int fd*, *struct statvfs *buf*);

int
fstatvfs1(*int fd*, *struct statvfs *buf*, *int flags*);

DESCRIPTION

statvfs() and **statvfs1**() return information about a mounted file system. *path* is the path name of any file within the mounted file system. *buf* is a pointer to a **statvfs** structure defined in statvfs(5).

fstatvfs() and **fstatvfs1**() return the same information about an open file referenced by descriptor *fd*.

The **statvfs1**() and **fstatvfs1**() functions allow an extra *flags* argument which can be ST_WAIT and ST_NOWAIT. When ST_NOWAIT is specified, then only cached statistics are returned. This can result in significant savings on non-local filesystems, where gathering statistics involves a network communication.

The **statvfs**() and **fstatvfs**() calls are equivalent to the respective **statvfs1**() and **fstatvfs1**() calls with ST_WAIT specified as the *flags* argument.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

statvfs() and **statvfs1**() fail if one or more of the following are true:

[ENOTDIR]	A component of the path prefix of *path* is not a directory.
[ENAMETOOLONG]	The length of a component of *path* exceeds NAME_MAX characters, or the length of *path* exceeds PATH_MAX characters.
[ENOENT]	The file referred to by *path* does not exist.
[EACCES]	Search permission is denied for a component of the path prefix of *path*.
[ELOOP]	Too many symbolic links were encountered in translating *path*.
[EFAULT]	*buf* or *path* points to an invalid address.
[EIO]	An I/O error occurred while reading from or writing to the file system.

fstatvfs() and **fstatvfs1**() fail if one or more of the following are true:

`[EBADF]`	*`fd`* is not a valid open file descriptor.
`[EFAULT]`	*`buf`* points to an invalid address.
`[EIO]`	An I/O error occurred while reading from or writing to the file system.

SEE ALSO

`statvfs`(5)

HISTORY

The **`statvfs`**(), **`statvfs1`**(), **`fstatvfs`**(), and **`fstatvfs1`**() functions first appeared in NetBSD 3.0 to replace the **`statfs`**() family of functions which first appeared in 4.4BSD.

NAME

swapctl — modify swap configuration

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>
#include <sys/swap.h>

int
swapctl(*int cmd*, *void *arg*, *int misc*);

DESCRIPTION

The **swapctl** function is used to add and delete swap devices, and modify their configuration.

The *cmd* parameter specifies the operation to be performed. The *arg* and *misc* parameters have different meanings, depending on the *cmd* parameter.

If *cmd* is SWAP_NSWAP, the current number of swap devices in the system is returned. The *arg* and *misc* parameters are ignored.

If *cmd* is SWAP_STATS, the current statistics for swap devices are returned in the *arg* parameter. No more than *misc* swap devices are returned. The *arg* parameter should point to an array of at least *misc* struct swapent structures:

```
struct swapent {
        dev_t   se_dev;                /* device id */
        int     se_flags;              /* entry flags */
        int     se_nblks;              /* total blocks */
        int     se_inuse;              /* blocks in use */
        int     se_priority;           /* priority */
        char    se_path[PATH_MAX+1];   /* path to entry */
};
```

The flags are defined as

```
        SWF_INUSE       in use: we have swapped here
        SWF_ENABLE      enabled: we can swap here
        SWF_BUSY        busy: I/O happening here
        SWF_FAKE        fake: still being built
```

If *cmd* is SWAP_ON, the *arg* parameter is used as a pathname of a file to enable swapping to. The *misc* parameter is used to set the priority of this swap device.

If *cmd* is SWAP_OFF, the *arg* parameter is used as the pathname of a file to disable swapping from. The *misc* parameter is ignored.

If *cmd* is SWAP_CTL, the *arg* and *misc* parameters have the same function as for the SWAP_ON case, except that they change the priority of a currently enabled swap device.

If *cmd* is SWAP_DUMPDEV, the *arg* parameter is used as the pathname of a device to use as the dump device, should the system panic.

If *cmd* is SWAP_GETDUMPDEV, the *arg* parameter points to a dev_t, which is filled in by the current dump device.

When swapping is enabled on a block device, the first portion of the disk is left unused to prevent any disklabel present from being overwritten. This space is allocated from the swap device when the `SWAP_ON` command is used.

The priority of a swap device can be used to fill faster swap devices before slower ones. A priority of 0 is the highest, with larger numbers having lower priority. For a fuller discussion on swap priority, see the **SWAP PRIORITY** section in `swapctl`(8).

RETURN VALUES

If the *`cmd`* parameter is `SWAP_NSWAP` or `SWAP_STATS`, **`swapctl`**() returns the number of swap devices, if successful. The `SWAP_NSWAP` command is always successful. Otherwise it returns 0 on success and −1 on failure, setting the global variable *errno* to indicate the error.

ERRORS

`swapctl`() succeeds unless:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded `NAME_MAX` characters, or an entire path name exceeded `PATH_MAX` characters.
[ENOENT]	The named device does not exist. For the `SWAP_CTL` command, the named device is not currently enabled for swapping.
[EACCES]	Search permission is denied for a component of the path prefix.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EPERM]	The caller is not the super-user.
[EBUSY]	The device specified by *`arg`* has already been made available for swapping.
[EINVAL]	The device configured by *`arg`* has no associated size, or the *`cmd`* was unknown.
[ENXIO]	The major device number of *`arg`* is out of range (this indicates no device driver exists for the associated hardware).
[ENXIO]	The block device specified by *`arg`* is not marked as a swap partition in the disklabel.
[EIO]	An I/O error occurred while opening the swap device.
[EFAULT]	*`arg`* points outside the process' allocated address space.

SEE ALSO

`swapctl`(8)

HISTORY

The **`swapctl`**() function call appeared in NetBSD 1.3. The *`se_path`* member was added to *struct swapent* in NetBSD 1.4, when the header file was also moved from ⟨`vm/vm_swap.h`⟩ to its current location in ⟨`sys/swap.h`⟩.

AUTHORS

The current swap system was designed and implemented by Matthew Green ⟨mrg@eterna.com.au⟩, with help from Paul Kranenburg ⟨pk@NetBSD.org⟩ and Leo Weppelman ⟨leo@NetBSD.org⟩, and insights from Jason R. Thorpe ⟨thorpej@NetBSD.org⟩.

NAME

symlink — make symbolic link to a file

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
symlink(*const char ∗name1*, *const char ∗name2*);

DESCRIPTION

A symbolic link *name2* is created to *name1* (*name2* is the name of the file created, *name1* is the string used in creating the symbolic link). Either name may be an arbitrary path name; the files need neither to be on the same file system nor to exist.

RETURN VALUES

Upon successful completion, a zero value is returned. If an error occurs, the error code is stored in *errno* and a −1 value is returned.

ERRORS

The symbolic link succeeds unless:

[ENOTDIR]	A component of the *name2* prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	A component of the *name2* path does not exist.
[EACCES]	A component of the *name2* path prefix denies search permission.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EEXIST]	*name2* already exists.
[EIO]	An I/O error occurred while making the directory entry for *name2*, or allocating the inode for *name2*, or writing out the link contents of *name2*.
[EROFS]	The file *name2* would reside on a read-only file system.
[ENOSPC]	The directory in which the entry for the new symbolic link is being placed cannot be extended because there is no space left on the file system containing the directory.
[ENOSPC]	The new symbolic link cannot be created because there there is no space left on the file system that will contain the symbolic link.
[ENOSPC]	There are no free inodes on the file system on which the symbolic link is being created.
[EDQUOT]	The directory in which the entry for the new symbolic link is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
[EDQUOT]	The new symbolic link cannot be created because the user's quota of disk blocks on the file system that will contain the symbolic link has been exhausted.

`[EDQUOT]`	The user's quota of inodes on the file system on which the symbolic link is being created has been exhausted.
`[EIO]`	An I/O error occurred while making the directory entry or allocating the inode.
`[EFAULT]`	*name1* or *name2* points outside the process's allocated address space.

SEE ALSO

`ln`(1), `link`(2), `readlink`(2), `unlink`(2), `symlink`(7)

HISTORY

The **symlink**() function call appeared in 4.2BSD.

NAME

sync — synchronize disk block in-core status with that on disk

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

void
sync(*void*);

DESCRIPTION

The **sync**() function forces a write of dirty (modified) buffers in the block buffer cache out to disk. The kernel keeps this information in core to reduce the number of disk I/O transfers required by the system. As information in the cache is lost after a system crash, kernel thread **ioflush** ensures that dirty buffers are synced to disk eventually. By default, a dirty buffer is synced after 30 seconds, but some filesystems exploit **ioflush** features to sync directory data and metadata faster (after 15 and 10 seconds, respectively).

The function `fsync`(2) may be used to synchronize individual file descriptor attributes.

SEE ALSO

`fsync`(2), `sync`(8)

HISTORY

A **sync**() function call appeared in Version 6 AT&T UNIX.

BUGS

sync() may return before the buffers are completely flushed.

NAME

sysarch — architecture-dependent system call

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <machine/sysarch.h>

int
sysarch(*int number*, *void *args*);

DESCRIPTION

sysarch() performs the architecture-dependent function specified by *number* with the arguments specified by the *args* pointer. *args* is a pointer to a structure defining the actual arguments of the function. Symbolic constants and argument structures for the architecture-dependent functions can be found in the header file ⟨machine/sysarch.h⟩.

The **sysarch**() system call should never be called directly by user programs. Instead, they should access its functions using the architecture-dependent library.

RETURN VALUES

See the manual pages for specific architecture-dependent function calls for information about their return values.

HISTORY

The **sysarch**() function call appeared in NetBSD 1.0.

NAME

syscall, **__syscall** — indirect system call

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/syscall.h>
#include <unistd.h>

int
syscall(*int number*, *...*);

quad_t
__syscall(*quad_t number*, *...*);

DESCRIPTION

syscall() performs the system call whose assembly language interface has the specified *number* with the specified arguments. Symbolic constants for system calls can be found in the header file 〈sys/syscall.h〉. The **__syscall** form should be used when one or more of the parameters is a 64-bit argument to ensure that argument alignment is correct. This system call is useful for testing new system calls that do not have entries in the C library.

RETURN VALUES

The return values are defined by the system call being invoked. In general, a 0 return value indicates success. A −1 return value indicates an error, and an error code is stored in *errno*.

HISTORY

The **syscall**() function call appeared in 4.0BSD.

BUGS

There is no way to simulate system calls that have multiple return values such as pipe(2).

Since architectures return 32 bit and 64 bit results in different registers, it may be impossible to portably convert the result of **__syscall**() to a 32bit value. For instance sparc returns 32 bit values in %o0 and 64 bit values in %o0:%o1 (with %o0 containing the most significant part) so a 32 bit right shift of the result is needed to get a correct 32 bit result.

Due to ABI implementation differences in passing struct or union type arguments to system calls between different processors, all system calls pass instead pointers to such structs or unions, even when the documentation of the system call mentions otherwise. The conversion between passing structs and unions is handled normally via userland stubs. The correct arguments for the kernel entry points for each system call can be found in the header file 〈sys/syscallargs.h〉

NAME

timer_create — create a per-process timer

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <time.h>
#include <signal.h>

int
timer_create(*clockid_t clockid*, *struct sigevent * restrict evp*,
    *timer_t * restrict timerid*);

DESCRIPTION

The **timer_create**() function creates a per-process timer using the clock specified in the *clockid* argument. If it succeeds, the **timer_create**() function fills in the *timerid* argument with an id associated with the timer created that can be used by other timer related calls. The *clockid* must be a valid clock id as defined in ⟨time.h⟩. The timer is created in a disarmed state.

An optional (non-NULL) sigevent argument can be specified by the *evp* argument. If the *evp* argument is NULL, then it defaults to *sigev_notify* set to SIGEV_SIGVAL and *sigev_value* set to *timerid*. See siginfo(2) for accessing those values from a signal handler.

NOTES

Timers are not inherited after a fork(2) and are disarmed and deleted by an exec(3).

RETURN VALUES

If successful, the **timer_create**() function returns 0, and fills in the *timerid* argument with the id of the new timer that was created. Otherwise, it returns −1, and sets errno to indicate the error.

ERRORS

The **timer_create**() function will fail if:

[EAGAIN]	The system is out of resources to satisfy this request, or the process has created all the timers it is allowed.
[EINVAL]	The argument *clockid* is not a valid clock id.

SEE ALSO

clock_getres(2), clock_gettime(2), clock_settime(2), timer_delete(2), timer_getoverrun(2), timer_gettime(2), timer_settime(2)

STANDARDS

IEEE Std 1003.1b-1993 (“POSIX.1”), IEEE Std 1003.1i-1995 (“POSIX.1”)

NAME

timer_delete — delete a per-process timer

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <time.h>

int
timer_delete(*timer_t timerid*);

DESCRIPTION

The **timer_delete**() functions deletes the timer specified in the *timerid* argument. The *timerid* argument must point to valid timer id, created by timer_create(2). If the deletion is successful, the timer is disarmed and deleted. Pending notification events (signals) may or may not be delivered.

RETURN VALUES

If successful, the **timer_delete**() functions returns 0. Otherwise, it returns −1, and sets errno to indicate error.

ERRORS

The **timer_delete**() function will fail if:

[EINVAL]	The argument *timerid* is not a valid timer id.

SEE ALSO

timer_create(2), timer_getoverrun(2), timer_gettime(2), timer_settime(2)

STANDARDS

IEEE Std 1003.1b-1993 (“POSIX.1”), IEEE Std 1003.1i-1995 (“POSIX.1”)

NAME

timer_settime, **timer_gettime**, **timer_getoverrun** — process timer manipulation

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <time.h>

int
timer_settime(*timer_t timerid*, *int flags*,
    const struct itimerspec ∗ restrict tim,
    struct itimerspec ∗ restrict otim);

int
timer_gettime(*timer_t timerid*, *struct itimerspec ∗tim*);

int
timer_getoverrun(*timer_t timerid*);

DESCRIPTION

The **timer_settime**() sets the next expiration time of the timer with id *timerid* to the *it_value* specified in the *tim* argument. If the value is 0, the timer is disarmed. If the argument *otim* is not NULL the old timer settingas are returned. If the *flags* argument is set to TIMER_RELTIME then the expiration time is set to the value in nanoseconds specified in the *tim* argument from the time the call to **timer_settime**() was made. If the *flags* argument is set to TIMER_ABSTIME then the expiration time is set to be equal to the difference between the clock associated with this timer, and the value specified in the *tim* argument. If that time has already passed, then the call succeeds, and the expiration notification occurs.

If the *it_interval* of the *tim* argument is non-zero, then the timer reloads upon expiration.

The **timer_gettime**() function returns the current settings of the timer specified by the *timerid* argument in the *tim* argument.

Only one notification event (signal) can be pending for a given timer and process. If a timer expires while the signal is still queued for delivery, then the overrun counter for that timer is increased. The counter can store values up to DELAYTIMER_MAX. When the signal is finally delivered to the process, then the **timer_getoverrun**() function can be used to retrieve the overrun counter for the timer specified in the *timerid* argument.

NOTES

Expiration time values are always rounded up to the resolution of the timer, so a notification will never be sent before the requested time. Values returned in the *otim* argument of **timer_settime**() or in the *tim* argment of **timer_gettime**() are subject to the above rounding effect and might not exactly match the requested values by the user.

RETURN VALUES

If successful, the **timer_gettime**() and **timer_settime**() functions return 0, and the **timer_getoverrun**() function returns the expiration overrun count for the specified timer. Otherwise, the functions return −1, and set errno to indicate the error.

ERRORS

The **timer_gettime**(), **timer_getoverrun**(), and **timer_settime**() functions will fail if:

[EINVAL]	The argument *timerid* does not correspond to a valid timer id as returned by **timer_create**() or that timer id has been deleted by **timer_delete**().

The **timer_settime**() function will fail if:

[EINVAL]	A nanosecond field in the *tim* structure specified a value less than zero or greater than or equal to 10e9.

SEE ALSO

clock_gettime(2), timer_create(2), timer_delete(2)

STANDARDS

IEEE Std 1003.1b-1993 (“POSIX.1”), IEEE Std 1003.1i-1995 (“POSIX.1”)

NAME

truncate, **ftruncate** — truncate a file to a specified length

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
truncate(*const char *path*, *off_t length*);

int
ftruncate(*int fd*, *off_t length*);

DESCRIPTION

truncate() causes the file named by *path* or referenced by *fd* to have a size of *length* bytes. If the file previously was larger than this size, the extra data is discarded. If it was previously shorter than *length*, its size is increased to the specified value and the extended area appears as if it were zero-filled.

With **ftruncate**(), the file must be open for writing; for **truncate**(), the process must have write permissions for the file.

RETURN VALUES

A value of 0 is returned if the call succeeds. If the call fails a −1 is returned, and the global variable *errno* specifies the error.

ERRORS

Error return codes common to **truncate**() and **ftruncate**() are:

[EISDIR]	The named file is a directory.
[EROFS]	The named file resides on a read-only file system.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed.
[EIO]	An I/O error occurred updating the inode.
[ENOSPC]	There was no space in the filesystem to complete the operation.

Error codes specific to **truncate**() are:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix, or the named file is not writable by the user.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EFAULT]	*path* points outside the process's allocated address space.

Error codes specific to **ftruncate**() are:

[EBADF]	The *fd* is not a valid descriptor.
[EINVAL]	The *fd* references a socket, not a file, or the *fd* is not open for writing.

SEE ALSO

open(2)

STANDARDS

Use of **truncate**() to extend a file is an IEEE Std 1003.1-2004 " ("POSIX.1") extension, and is thus not portable. Files can be extended in a portable way seeking (using lseek(2)) to the required size and writing a single character with write(2).

HISTORY

The **truncate**() and **ftruncate**() function calls appeared in 4.2BSD.

BUGS

These calls should be generalized to allow ranges of bytes in a file to be discarded.

NAME

ucontext — user context

SYNOPSIS

#include <ucontext.h>

DESCRIPTION

ucontext_t is a structure type which is used to describe the context of a thread of control within the execution of a process.

ucontext_t includes the following members:

```
ucontext_t *  uc_link
sigset_t      uc_sigmask
stack_t       uc_stack
mcontext_t    uc_mcontext
```

The *uc_link* member points to the context that will be resumed after the function specified when modifying a context using makecontext(3) has returned. If *uc_link* is a null pointer, then the context is the main context, and the process will exit with an exit status of 0 upon return.

The *uc_sigmask* member is the set of signals that are blocked when the context is activated. Further information can be found in sigprocmask(2).

The *uc_stack* member defines the stack used by the context. Further information can be found in sigaltstack(2).

The *uc_mcontext* member defines the machine state associated with the context; it may consist of general registers, floating point registers and other machine-specific information. Its description is beyond the scope of this manual page; portable applications should not access this structure member.

SEE ALSO

_exit(2), getcontext(2), setcontext(2), sigaltstack(2), sigprocmask(2), makecontext(3), swapcontext(3)

STANDARDS

The **ucontext_t** type conforms to X/Open System Interfaces and Headers Issue 5 (“XSH5”).

HISTORY

The **makecontext**() and **swapcontext**() functions first appeared in AT&T System V.4 UNIX.

NAME

umask — set file creation mode mask

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/stat.h>

mode_t
umask(*mode_t numask*);

DESCRIPTION

The **umask**() routine sets the process's file mode creation mask to *numask* and returns the previous value of the mask. The 9 low-order access permission bits of *numask* are used by system calls, including open(2), mkdir(2), mkfifo(2) and mknod(2) to turn off corresponding bits requested in file mode. (See chmod(2)). This clearing allows each user to restrict the default access to his files.

The default mask value is S_IWGRP|S_IWOTH (022, write access for the owner only). Child processes inherit the mask of the calling process.

RETURN VALUES

The previous value of the file mode mask is returned by the call.

ERRORS

The **umask**() function is always successful.

SEE ALSO

chmod(2), mkdir(2), mkfifo(2), mknod(2), open(2)

STANDARDS

The **umask**() function conforms to ISO/IEC 9945-1:1990 ("POSIX.1").

NAME

undelete — attempt to recover a deleted file

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
undelete(*const char *path*);

DESCRIPTION

The **undelete**() function attempts to recover the deleted file named by *path*. Currently, this works only when the named object is a whiteout in a union filesystem. The system call removes the whiteout causing any objects in a lower layer of the union stack to become visible once more.

Eventually, the **undelete** functionality may be expanded to other filesystems able to recover deleted files such as the log-structured filesystem.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

The **undelete**() succeeds unless:

[ENOTDIR]	A component of the path prefix is not a directory.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.
[EEXIST]	The path does not reference a whiteout.
[ENOENT]	The named whiteout does not exist.
[EACCES]	Search permission is denied for a component of the path prefix, or write permission is denied on the directory containing the name to be undeleted.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EPERM]	The directory containing the name is marked sticky, and the containing directory is not owned by the effective user ID.
[EIO]	An I/O error occurred while updating the directory entry.
[EROFS]	The name resides on a read-only file system.
[EFAULT]	*path* points outside the process's allocated address space.

SEE ALSO

unlink(2), mount_union(8)

HISTORY

An **undelete** function call first appeared in 4.4BSD -Lite.

NAME

unlink — remove directory entry

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

int
unlink(*const char *path*);

DESCRIPTION

The **unlink**() function removes the link named by *path* from its directory and decrements the link count of the file which was referenced by the link. If that decrement reduces the link count of the file to zero, and no process has the file open, then all resources associated with the file are reclaimed. If one or more process have the file open when the last link is removed, the link is removed, but the removal of the file is delayed until all references to it have been closed.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

The **unlink**() succeeds unless:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix, or write permission is denied on the directory containing the link to be removed.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EPERM]	The named file is a directory and the effective user ID of the process is not the super-user, the file system containing the file does not permit the use of **unlink**() on a directory, or the directory containing the file is marked sticky, and neither the containing directory nor the file to be removed are owned by the effective user ID.
[EBUSY]	The entry to be unlinked is the mount point for a mounted file system.
[EIO]	An I/O error occurred while deleting the directory entry or deallocating the inode.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*path* points outside the process's allocated address space.

SEE ALSO

close(2), link(2), rmdir(2), symlink(7)

STANDARDS

The **unlink**() function conforms to ISO/IEC 9945-1:1990 ("POSIX.1").

HISTORY

An **unlink**() function call appeared in Version 6 AT&T UNIX.

NAME

utimes, **lutimes**, **futimes** — set file access and modification times

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/time.h>

int
utimes(*const char *path*, *const struct timeval times[2]*);

int
lutimes(*const char *path*, *const struct timeval times[2]*);

int
futimes(*int fd*, *const struct timeval times[2]*);

DESCRIPTION

The access and modification times of the file named by *path* or referenced by *fd* are changed as specified by the argument *times*.

If *times* is NULL, the access and modification times are set to the current time. The caller must be the owner of the file, have permission to write the file, or be the super-user.

If *times* is non-NULL, it is assumed to point to an array of two timeval structures. The access time is set to the value of the first element, and the modification time is set to the value of the second element. The caller must be the owner of the file or be the super-user.

In either case, the inode-change-time of the file is set to the current time.

lutimes() is like **utimes**() except in the case where the named file is a symbolic link, in which case **lutimes**() changes the access and modification times of the link, while **utimes**() changes the times of the file the link references.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

utimes() and **lutimes**() will fail if:

[EACCES]	Search permission is denied for a component of the path prefix; or the *times* argument is NULL and the effective user ID of the process does not match the owner of the file, and is not the super-user, and write access is denied.
[EFAULT]	*path* or *times* points outside the process's allocated address space.
[EIO]	An I/O error occurred while reading or writing the affected inode.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
[ENOENT]	The named file does not exist.

[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The *times* argument is not NULL and the calling process's effective user ID does not match the owner of the file and is not the super-user.
[EROFS]	The file system containing the file is mounted read-only.

futimes() will fail if:

[EBADF]	*fd* does not refer to a valid descriptor.
[EACCES]	The *times* argument is NULL and the effective user ID of the process does not match the owner of the file, and is not the super-user, and write access is denied.
[EFAULT]	*times* points outside the process's allocated address space.
[EIO]	An I/O error occurred while reading or writing the affected inode.
[EPERM]	The *times* argument is not NULL and the calling process's effective user ID does not match the owner of the file and is not the super-user.
[EROFS]	The file system containing the file is mounted read-only.

SEE ALSO

stat(2), utime(3), symlink(7)

HISTORY

The **utimes**() function call appeared in 4.2BSD. The **futimes**() function call appeared in NetBSD 1.2. The **lutimes**() function call appeared in NetBSD 1.3.

NAME

utrace — insert user record in ktrace log

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/time.h>
#include <sys/uio.h>
#include <sys/ktrace.h>

int
utrace(const char *label, void *addr, size_t len);
```

DESCRIPTION

Adds a record to the process trace with information supplied by user. The record is identified by *label* and contains *len* bytes from memory pointed to by *addr*. This call only has an effect if the calling process is being traced.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

[ENOSYS]	Currently running kernel was compiled without ktrace(2) support (option KTRACE).
[EINVAL]	Specified data length *len* was bigger than KTR_USER_MAXLEN.

SEE ALSO

kdump(1), ktrace(1), ktruss(1), fktrace(2), ktrace(2), options(4)

HISTORY

The **utrace**() system call first appeared in FreeBSD 2.2. It was added to NetBSD in NetBSD 1.6. The *label* argument is a NetBSD extension.

NAME

uuidgen — generate universally unique identifiers

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/uuid.h>

int
uuidgen(*struct uuid *store*, *int count*);

DESCRIPTION

The **uuidgen**() system call generates *count* universally unique identifiers (UUIDs) and writes them to the buffer pointed to by *store*. The identifiers are generated according to the syntax and semantics of the DCE version 1 variant of universally unique identifiers. See below for a more in-depth description of the identifiers. When no IEEE 802 address is available for the node field, a random multi-cast address is generated for each invocation of the system call. According to the algorithm of generating time-based UUIDs, this will also force a new random clock sequence, thereby increasing the likelihood for the identifier to be unique.

When multiple identifiers are to be generated, the **uuidgen**() system call will generate a set of identifiers that is dense in such a way that there is no identifier that is larger than the smallest identifier in the set and smaller than the largest identifier in the set and that is not already in the set.

Universally unique identifiers, also known as globally unique identifiers (GUIDs), have a binary representation of 128-bits. The grouping and meaning of these bits is described by the following structure and its description of the fields that follow it:

```
struct uuid {
        uint32_t        time_low;
        uint16_t        time_mid;
        uint16_t        time_hi_and_version;
        uint8_t         clock_seq_hi_and_reserved;
        uint8_t         clock_seq_low;
        uint8_t         node[_UUID_NODE_LEN];
};
```

time_low	The least significant 32 bits of a 60-bit timestamp. This field is stored in the native byte-order.
time_mid	The least significant 16 bits of the most significant 28 bits of the 60-bit timestamp. This field is stored in the native byte-order.
time_hi_and_version	The most significant 12 bits of the 60-bit timestamp multiplexed with a 4-bit version number. The version number is stored in the most significant 4 bits of the 16-bit field. This field is stored in the native byte-order.
clock_seq_hi_and_reserved	The most significant 6 bits of a 14-bit sequence number multiplexed with a 2-bit variant value. Note that the width of the variant value is determined by the variant itself. Identifiers generated by the **uuidgen**() system call have variant value 10b. the variant value is stored in the most significant bits of the field.

clock_seq_low	The least significant 8 bits of a 14-bit sequence number.
node	The 6-byte IEEE 802 (MAC) address of one of the interfaces of the node. If no such interface exists, a random multi-cast address is used instead.

The binary representation is sensitive to byte ordering. Any multi-byte field is to be stored in the local or native byte-order and identifiers must be converted when transmitted to hosts that do not agree on the byte-order. The specification does not however document what this means in concrete terms and is otherwise beyond the scope of this system call.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **uuidgen**() system call can fail with:

`[EFAULT]`	The buffer pointed to by *store* could not be written to for any or all identifiers.
`[EINVAL]`	The *count* argument is less than 1 or larger than the hard upper limit of 2048.

SEE ALSO

uuidgen(1), uuid(3)

STANDARDS

The identifiers are represented and generated in conformance with the DCE 1.1 RPC specification. The **uuidgen**() system call is itself not part of the specification.

HISTORY

The **uuidgen**() system call first appeared in FreeBSD 5.0 and was subsequently added to NetBSD 2.0.

NAME

vfork — spawn new process in a virtual memory efficient way

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

pid_t
vfork(*void*);

DESCRIPTION

The **vfork** system call creates a new process that does not have a new virtual address space, but rather shares address space with the parent, thus avoiding potentially expensive copy-on-write operations normally associated with creating a new process. It is useful when the purpose of fork(2) would have been to create a new system context for an execve(2). The **vfork** system call differs from fork(2) in that the child borrows the parent's memory and thread of control until a call to execve(2) or an exit (either by a call to _exit(2) or abnormally). The parent process is suspended while the child is using its resources.

The **vfork** system call returns 0 in the child's context and (later) the pid of the child in the parent's context.

The **vfork** system call can normally be used just like fork(2). It does not work, however, to return while running in the childs context from the procedure that called **vfork**() since the eventual return from **vfork**() would then return to a no longer existent stack frame. Be careful, also, to call _exit(2) rather than exit(3) if you can't execve(2), since exit(3) will flush and close standard I/O channels, and thereby mess up the standard I/O data structures in the parent process. (Even with fork(2) it is wrong to call exit(3) since buffered data would then be flushed twice.)

RETURN VALUES

Same as for fork(2).

ERRORS

Same as for fork(2).

SEE ALSO

execve(2), fork(2), sigaction(2), wait(2)

HISTORY

The **vfork**() function call appeared in 3.0BSD. In 4.4BSD, the semantics were changed to only suspend the parent. The original semantics were reintroduced in NetBSD 1.4.

BUGS

Users should not depend on the memory sharing semantics of **vfork**() as other ways of speeding up the fork process may be developed in the future.

To avoid a possible deadlock situation, processes that are children in the middle of a **vfork**() are never sent SIGTTOU or SIGTTIN signals; rather, output or ioctl(2) calls are allowed and input attempts result in an end-of-file indication.

NAME

wait, **waitpid**, **wait4**, **wait3** — wait for process termination

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <sys/wait.h>

pid_t
wait(*int *status*);

pid_t
waitpid(*pid_t wpid*, *int *status*, *int options*);

#include <sys/resource.h>

pid_t
wait3(*int *status*, *int options*, *struct rusage *rusage*);

pid_t
wait4(*pid_t wpid*, *int *status*, *int options*, *struct rusage *rusage*);

DESCRIPTION

The **wait**() function suspends execution of its calling process until *status* information is available for a terminated child process, or a signal is received. On return from a successful **wait**() call, the *status* area contains termination information about the process that exited as defined below.

The **wait4**() call provides a more general interface for programs that need to wait for certain child processes, that need resource utilization statistics accumulated by child processes, or that require options. The other wait functions are implemented using **wait4**().

The *wpid* parameter specifies the set of child processes for which to wait. If *wpid* is −1, the call waits for any child process. If *wpid* is 0, the call waits for any child process in the process group of the caller. If *wpid* is greater than zero, the call waits for the process with process id *wpid*. If *wpid* is less than −1, the call waits for any process whose process group id equals the absolute value of *wpid*.

The *status* parameter is defined below.

The *options* parameter contains the bitwise OR of any of the following options:

WNOHANG This option is used to indicate that the call should not block if there are no processes that wish to report status.

WUNTRACED If this option is set, children of the current process that are stopped due to a SIGTTIN, SIGTTOU, SIGTSTP, or SIGSTOP signal also have their status reported.

WALTSIG If this option is specified, the call will wait only for processes that are configured to post a signal other than SIGCHLD when they exit. If WALTSIG is not specified, the call will wait only for processes that are configured to post SIGCHLD.

__WCLONE This is an alias for WALTSIG. It is provided for compatibility with the Linux clone(2) API.

WALLSIG If this option is specified, the call will wait for all children regardless of what exit signal they post.

__WALL This is an alias for WALLSIG. It is provided for compatibility with the Linux clone(2) API .

If *rusage* is non-zero, a summary of the resources used by the terminated process and all its children is returned (this information is currently not available for stopped processes).

When the `WNOHANG` option is specified and no processes wish to report status, **wait4**() returns a process id of 0.

The **waitpid**() call is identical to **wait4**() with an *rusage* value of zero. The older **wait3**() call is the same as **wait4**() with a *wpid* value of −1.

The following macros may be used to test the manner of exit of the process. Note that these macros expect the *status* value itself, not a pointer to the *status* value. One of the first three macros will evaluate to a non-zero (true) value:

WIFEXITED(*status*)
: True if the process terminated normally by a call to `_exit`(2) or `exit`(3).

WIFSIGNALED(*status*)
: True if the process terminated due to receipt of a signal.

WIFSTOPPED(*status*)
: True if the process has not terminated, but has stopped and can be restarted. This macro can be true only if the wait call specified the `WUNTRACED` option or if the child process is being traced (see `ptrace`(2)).

Depending on the values of those macros, the following macros produce the remaining status information about the child process:

WEXITSTATUS(*status*)
: If **WIFEXITED**(*status*) is true, evaluates to the low-order 8 bits of the argument passed to `_exit`(2) or `exit`(3) by the child.

WTERMSIG(*status*)
: If **WIFSIGNALED**(*status*) is true, evaluates to the number of the signal that caused the termination of the process.

WCOREDUMP(*status*)
: If **WIFSIGNALED**(*status*) is true, evaluates as true if the termination of the process was accompanied by the creation of a core file containing an image of the process when the signal was received.

WSTOPSIG(*status*)
: If **WIFSTOPPED**(*status*) is true, evaluates to the number of the signal that caused the process to stop.

NOTES

See `sigaction`(2) for a list of termination signals. A status of 0 indicates normal termination.

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes are assigned the parent process 1 ID (the init process ID).

If a signal is caught while any of the **wait**() calls is pending, the call may be interrupted or restarted when the signal-catching routine returns, depending on the options in effect for the signal; see `intro`(2), System call restart.

RETURN VALUES

If **wait**() returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

If **wait4**(), **wait3**() or **waitpid**() returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. If there are no children not previously awaited, −1 is returned with *errno* set to [ECHILD]. Otherwise, if WNOHANG is specified and there are no stopped or exited children, 0 is returned. If an error is detected or a caught signal aborts the call, a value of −1 is returned and *errno* is set to indicate the error.

ERRORS

wait() will fail and return immediately if:

[ECHILD]	The calling process has no existing unwaited-for child processes.
[EFAULT]	The *status* or *rusage* arguments point to an illegal address. (May not be detected before exit of a child process.)
[EINTR]	The call was interrupted by a caught signal, or the signal did not have the SA_RESTART flag set.

In addition, **wait3**(), **wait4**(), and **waitpid**() will fail and return immediately if:

[EINVAL]	An invalid value was specified for *options*.

SEE ALSO

_exit(2), sigaction(2)

STANDARDS

The **wait**() and **waitpid**() functions conform to ISO/IEC 9945-1:1990 (“POSIX.1”); the **wait3**() function conforms to X/Open Portability Guide Issue 4 (“XPG4”); **wait4**() is an extension. The **WCOREDUMP**() macro and the ability to restart a pending **wait**() call are extensions to the POSIX interface.

HISTORY

A **wait**() function call appeared in Version 6 AT&T UNIX.

NAME

write, **writev**, **pwrite**, **pwritev** — write output

LIBRARY

Standard C Library (libc, −lc)

SYNOPSIS

#include <unistd.h>

ssize_t
write(*int d*, *const void *buf*, *size_t nbytes*);

ssize_t
pwrite(*int d*, *const void *buf*, *size_t nbytes*, *off_t offset*);

#include <sys/uio.h>

ssize_t
writev(*int d*, *const struct iovec *iov*, *int iovcnt*);

ssize_t
pwritev(*int d*, *const struct iovec *iov*, *int iovcnt*, *off_t offset*);

DESCRIPTION

write() attempts to write *nbytes* of data to the object referenced by the descriptor *d* from the buffer pointed to by *buf*. **writev**() performs the same action, but gathers the output data from the *iovcnt* buffers specified by the members of the *iov* array: iov[0], iov[1], ..., iov[iovcnt - 1]. **pwrite**() and **pwritev**() perform the same functions, but write to the specified position in the file without modifying the file pointer.

For **writev**() and **pwritev**(), the *iovec* structure is defined as:

```
struct iovec {
        void *iov_base;
        size_t iov_len;
};
```

Each *iovec* entry specifies the base address and length of an area in memory from which data should be written. **writev**() and **pwritev**() will always write a complete area before proceeding to the next.

On objects capable of seeking, the **write**() starts at a position given by the pointer associated with *d* (see lseek(2)). Upon return from **write**(), the pointer is incremented by the number of bytes which were written.

Objects that are not capable of seeking always write from the current position. The value of the pointer associated with such an object is undefined.

If the real user is not the super-user, then **write**() clears the set-user-id bit on a file. This prevents penetration of system security by a user who “captures” a writable set-user-id file owned by the super-user.

If **write**() succeeds it will update the st_ctime and st_mtime fields of the file’s meta-data (see stat(2)).

When using non-blocking I/O on objects such as sockets that are subject to flow control, **write**() and **writev**() may write fewer bytes than requested; the return value must be noted, and the remainder of the operation should be retried when possible.

RETURN VALUES

Upon successful completion the number of bytes which were written is returned. Otherwise a −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

write(), **writev**(), **pwrite**(), and **pwritev**() will fail and the file pointer will remain unchanged if:

[EBADF]	*d* is not a valid descriptor open for writing.
[EPIPE]	An attempt is made to write to a pipe that is not open for reading by any process.
[EPIPE]	An attempt is made to write to a socket of type SOCK_STREAM that is not connected to a peer socket.
[EFBIG]	An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.
[EFAULT]	Part of *iov* or data to be written to the file points outside the process's allocated address space.
[EINVAL]	The pointer associated with *d* was negative.
[EINVAL]	The total length of the I/O is more than can be expressed by the ssize_t return value.
[ENOSPC]	There is no free space remaining on the file system containing the file.
[EDQUOT]	The user's quota of disk blocks on the file system containing the file has been exhausted.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[EINTR]	A signal was received before any data could be written to a slow device. See sigaction(2) for more information on the interaction between signals and system calls.
[EAGAIN]	The file was marked for non-blocking I/O, and no data could be written immediately.

In addition, **writev**() and **pwritev**() may return one of the following errors:

[EINVAL]	*iovcnt* was less than or equal to 0, or greater than {IOV_MAX}.
[EINVAL]	One of the *iov_len* values in the *iov* array was negative.
[EINVAL]	The sum of the *iov_len* values in the *iov* array overflowed a 32-bit integer.

pwrite() and **pwritev**() calls may also return the following errors:

[EINVAL]	The specified file offset is invalid.
[ESPIPE]	The file descriptor is associated with a pipe, socket, or FIFO.

SEE ALSO

fcntl(2), lseek(2), open(2), pipe(2), poll(2), select(2), sigaction(2)

STANDARDS

The **write**() function is expected to conform to IEEE Std 1003.1-1988 ("POSIX.1"). The **writev**() and **pwrite**() functions conform to X/Open Portability Guide Issue 4, Version 2 ("XPG4.2").

HISTORY

The **pwritev**() function call appeared in NetBSD 1.4. The **pwrite**() function call appeared in AT&T System V.4 UNIX. The **writev**() function call appeared in 4.2BSD. The **write**() function call appeared in

Version 6 AT&T UNIX.

CAVEATS

Error checks should explicitly test for −1. Code such as

```
while ((nr = write(fd, buf, sizeof(buf))) > 0)
```

is not maximally portable, as some platforms allow for *nbytes* to range between `SSIZE_MAX` and `SIZE_MAX` − 2, in which case the return value of an error-free **`write`**() may appear as a negative number distinct from −1. Proper loops should use

```
while ((nr = write(fd, buf, sizeof(buf))) != -1 && nr != 0)
```

NAME

x86_64_get_mtrr, **x86_64_set_mtrr** — access Memory Type Range Registers

LIBRARY

x86_64 Architecture Library (libx86_64, −lx86_64)

SYNOPSIS

#include <sys/types.h>
#include <machine/sysarch.h>
#include <machine/mtrr.h>

int
x86_64_get_mtrr(*struct mtrr ∗mtrrp*, *int ∗n*);

int
x86_64_set_mtrr(*struct mtrr ∗mtrrp*, *int ∗n*);

DESCRIPTION

These functions provide an interface to the MTRR registers found on 686-class processors for controlling processor access to memory ranges. This is most useful for accessing devices such as video accelerators on pci(4) and agp(4) buses. For example, enabling write-combining allows bus-write transfers to be combined into a larger transfer before bursting over the bus. This can increase performance of write operations 2.5 times or more.

mtrrp is a pointer to one or more mtrr structures, as described below. The *n* argument is a pointer to an integer containing the number of structures pointed to by *mtrrp*. For **x86_64_set_mtrr**() the integer pointed to by *n* will be updated to reflect the actual number of MTRRs successfully set. For **x86_64_get_mtrr**() no more than *n* structures will be copied out, and the integer value pointed to by *n* will be updated to reflect the actual number of valid structures retrieved. A NULL argument to *mtrrp* will result in just the number of MTRRs available being returned in the integer pointed to by *n*.

The argument *mtrrp* has the following structure:

```
struct mtrr {
        uint64_t base;
        uint64_t len;
        uint8_t type;
        int flags;
        pid_t owner;
};
```

The location of the mapping is described by its physical base address *base* and length *len*. Valid values for *type* are:

MTRR_TYPE_UC	uncached memory
MTRR_TYPE_WC	use write-combining
MTRR_TYPE_WT	use write-through caching
MTRR_TYPE_WP	write-protected memory
MTRR_TYPE_WB	use write-back caching

Valid values for *flags* are:

MTRR_PRIVATE
own range, reset the MTRR when the current process exits

MTRR_FIXED	use fixed range MTRR
MTRR_VALID	entry is valid

The *owner* member is the PID of the user process which claims the mapping. It is only valid if MTRR_PRIVATE is set in *flags*. To clear/reset MTRRs, use a *flags* field without MTRR_VALID set.

RETURN VALUES

Upon successful completion zero is returned, otherwise −1 is returned on failure, and the global variable *errno* is set to indicate the error. The integer value pointed to by *`n`* will contain the number of successfully processed mtrr structures in both cases.

ERRORS

`[ENOSYS]`	The currently running kernel or CPU has no MTRR support.
`[EINVAL]`	The currently running kernel has no MTRR support, or one of the mtrr structures pointed to by *`mtrrp`* is invalid.
`[EBUSY]`	No unused MTRRs are available.

HISTORY

The **`x86_64_get_mtrr`**() and **`x86_64_set_mtrr`**() were derived from their i386 counterparts, which appeared in NetBSD 1.6.

NAME

x86_64_iopl — change the x86_64 I/O privilege level

LIBRARY

x86_64 Architecture Library (libx86_64, −lx86_64)

SYNOPSIS

```
#include <sys/types.h>
#include <machine/sysarch.h>

int
x86_64_iopl(int iopl);
```

DESCRIPTION

x86_64_iopl() sets the x86_64 I/O privilege level to the value specified by *iopl*. This call is restricted to the super-user.

RETURN VALUES

Upon successful completion, **x86_64_iopl**() returns 0. Otherwise, a value of −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

x86_64_iopl() will fail if:

[EPERM] The caller was not the super-user, or the operation was not permitted at the current security level.

WARNING

You can really hose your machine if you enable user-level I/O and write to hardware ports without care.